Broadview 计算机专业人员书库 MK

计算机图形学几何工具算法详解

Geometric Tools for Computer Graphics

[美] Philip J.Schneider David H.Eberly 著

周长发 译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

VISIT…

LANZAROTE Caliente.com

Broadview 技术凝聚实力 · 专业创新出版

程序设计人员工作一个小时的价值通常高于本书的价格。从这个意义来说，你拥有了一本潜在价值为数千美元的书籍。花费这一价值的一小部分即可购得本书，这真是一个**现代的奇迹**。本书包含了令人难以置信的信息量！

——Eric Haines

本书特色

- 包含了大量健壮的算法，将节省你的时间，并帮你避免代价昂贵的错误。
- 涵盖了与二维和三维图形编程相关的各种问题。
- 每一个问题和解决方法都是独立论述的，只要阅读你需要的章节，就能得到所需的完整内容。
- 提供了帮你理解算法并将其用于实际工作所需的数学和几何背景知识。
- 清晰地图示每一个问题，并用易于理解的伪码来表示各种算法。

可在网站 www.mkp.com/gtcg 获得与本书相关的各种资源。

图书分类：计算机理论 > 图形学

ISBN 7-121-00515-8
9 787121 005152 >

网上订购：www.dearbook.com.cn
第一书店 · 第一服务

责任编辑：朱沭红　高洪霞
封面设计：张子健　李云霞

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书

ISBN 7-121-00515-8　定价：89.00元

计算机专业人员书库

计算机图形学几何工具算法详解

Geometric Tools for Computer Graphics

[美] Philip J. Schneider
David H. Eberly 著

周长发 译

Publishing House of Electronics Industry
北京 • BEIJING

内 容 简 介

本书提供了计算机图形学基础问题的各种有效算法，以及相关的数学和几何背景知识，对计算机图形学和其他领域的二维和三维几何学问题进行了全面的解析和合理的组织。本书包括建立基础图元、距离计算、近似值处理、包含性分析、分解、相交确定、分离等方面的算法，对每一个问题都有清晰的论述和图示，并利用易于理解的伪码来表示各种完整详尽的算法。除此之外，本书还在多个附录中提供了丰富的参考资料。

本书适合作为计算机图形学几何算法课程的教材，也可作为参考指南，供经验丰富的业界人士参考查阅。

Copyright © 2002 by Elsevier Science (USA).
Translation Copyright © 2004 by Publishing House of Electronics Industry.
All rights reserved.

本书中文简体版专有出版权由 Elsevier Inc. 授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2003-3945

图书在版编目（CIP）数据

计算机图形学几何工具算法详解／（美）史奈德（Schneider, P. J.）等著；周长发译. —北京：电子工业出版社，2005.1
（计算机专业人员书库）
书名原文：Geometric Tools for Computer Graphics
ISBN 7-121-00515-8

I. 计… II. ①史… ②周… III. 计算几何—应用—计算机图形学 IV. TP391.41
中国版本图书馆 CIP 数据核字（2004）第 123407 号

责任编辑：朱沭红　高洪霞
印　　刷：北京东光印刷厂
出版发行：电子工业出版社
　　　　　北京市海淀区万寿路 173 信箱　邮编 100036
经　　销：各地新华书店
开　　本：787×1092　1/16　印张：47.5　字数：1091 千字
印　　次：2005 年 1 月第 1 次印刷
印　　数：4000 册　定价：89.00 元

凡购买电子工业出版社的图书，如有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系。联系电话：（010）68279077。质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

献给我的妻子 Suzanne，我的儿子 Dakota 和 Jordan ——PS

献给我的妻子 Shelly，感谢她在我又写作一本书

的过程中所呈现的耐心 ——DE

关于作者

Philip Schneider 在沃特迪斯尼特色动画公司领导一个建模和动态模拟软件小组。在此之前，他曾在苹果公司和数字设备公司从事三维图形研究，研究领域涉及从底层接口到图形库和交互式应用的多个方面。他在华盛顿大学获得计算机科学专业的硕士学位。

David Eberly 是魔法软件公司的总裁，也是实时三维游戏引擎“疯狂魔法”的设计师。在此之前，他曾是数值设计有限公司的工程总监，该公司发行了实时三维游戏引擎 NetImmerse 他在北卡罗莱纳州大学 Chapel Hill 分校获得计算机科学专业的博士学位，在科罗拉多大学 Boulder 分校获得数学专业的博士学位。

序

我的书架上有一本名为《程序员的几何学》（*A Programmer's Geometry*）的老书，作者是 Bowyer 和 Woodwark。该书初版于 1983 年，1984 年和 1985 年再版两次后未再重印。多年来，我一直珍藏该书，仔细地记录谁借阅了它。我最近在万维网上搜索该书，结果发现 6 本旧书，它们的价格在 50 美元～100 美元之间。对于一本只有 140 页的平装书来说，这样的价格是非常高的。这本书能值那么多钱，是因为它说明了如何编程实现与二维几何学相关的各种操作。它不仅介绍了几何公式，而且还描述了在程序中实现这些公式的有效方法，并且提供了代码片断（用 FORTRAN 编写的代码片断）。

现在，几乎过去了 20 年，我们发现了一本新书，堪当这本薄书的继承者。呈现在你面前的这本书从浩如烟海的几何学文献中萃取了对程序设计人员最为有用的内容。自 1993 年以来，计算机图形学领域已经有了很大的发展，本书反映了这些巨变。受惠于计算机处理器性能的持续改善，过去需要脱机运行的许多运算，现在一般都可在交互程序中完成。多边形的三角剖分、碰撞检测和响应，以及曲面建模和修正，现在都可能按实时的速率完成。本书提供了实现上述及其他许多算法的坚实理论和代码。

除了提供与几何学相关的各种不同领域的任务的可靠参考，本书还介绍了隐藏在算法后面的基础理论。与采用纯粹的菜谱方式（这种方式仅让读者留下可运行的代码，却难以让读者理解代码是如何工作的）的一般书籍不同，作者在本书中解释了一系列关键的概念，这对读者是非常有益的。本书的阐述方法使每一个算法都能成为一种能进一步与其他工具重组的工具。

计算机图形学具有不断动态发展的特性，这使它成为一个特别令人感兴趣的学习领域。着色方法的研究和实现随着所使用硬件的不断发展而发展。例如，在交互式着色领域中，图形加速器的可编程特性一出现，就已经改变了不同技术的相对成本。更通俗地说，CPU 的发展已经使内存访问和高速缓存显得比过去用减少运算符次数（例如，计算乘法和加法）的方法来提高运算速度更为重要。然而，基础的理论和算法，比如得到物体凸面外壳的算法就相当经典，受这类变化的影响较小。当然，更加有效的算法总是在不断被发现，而且哪种方法被认为是当前最快的算法还受硬件的影响，但是基本的原理保持不变。因而，在把有关 DirectX 9 和 Intel 的 64 位 Itanium 体系结构的书籍从书架上撤下多年以后，你很可能还会在书架上保留本书的一些版本。

本书的价值之所以将持续存在，还因为有因特网。本人是"图形珍宝（Graphics Gems）"系列书籍代码库的管理者。在这一系列书籍于 20 世纪 90 年代早期出版时，这系列书籍的代码，包括 Philip Schneider 编写的代码，很明智地免费提供给读者使用。这些年来，读者不断给代码库寄来缺陷修正和代码改进，使所有用户从中受益。同样地，Dave Eberly 一直认真地维护他的"魔法软件（Magic Software）"网站（www.magic-software.com），其中包含了本书提供的许多算法的工作版本。计算机图形学领域的一位前沿研究者认为该网站是"国宝"，在任何需要的时候这一网站会立刻增加附录和勘误信息。代码不会过时，而是会在有效的支持下不断改进。本书中的算法更是如此，因为它们与特定硬件、网络协议或

其他短暂的事物并不相关。

多年来，我和其他的许多人已经在产品和研究项目中使用了作者们编写的算法和代码。程序设计人员工作一个小时的价值通常高于本书的价格。从这个意义来说，你拥有了一本潜在价值为数千美元的书籍。花费这一价值的一小部分即可购得本书，这真是一个现代的奇迹。本书包含了令人难以置信的信息量。数学语言可能降低你的阅读速度，然而如果使用其他的描述方法替换，则必然会包含赘言，且对有些算法可能会描述得不是很精确。如果你想寻找能使阅读更轻松的描述方式，请继续搜索。就像想获得世界上大部分有价值的东西一样，完全地理解本书的内容，是需要付出努力和奋斗的。

Eric Haines

前 言

快速而廉价的大众化图形硬件的出现，已经引起了人们对计算机图形学知识日渐热切的渴求，人们希望了解在计算机游戏、科学可视化、医学图像分析、仿真和虚拟世界等应用程序中如何编程实现各种各样的几何任务。各种应用程序的类型没有改变，但应用方向却利用技术的优势不断发展（Crawford 2002）。新的应用方向甚至已经包括以代码分析和可视化调试为目的的三维环境，以及政党联合形式的分析，即将政党的信念表示为不同的凸体，它们之间的相交表明潜在的联合性。

在书籍、Web 站点、新闻组、期刊文章或行业杂志中，均可以找到许多关于图形的知识。有时这类资源容易理解，但更多的时候却显得晦涩难懂。有时它们提供了足够多的细节来阐明潜在的原理，但有时却没有提供细节。有时提出一个概念时考虑到了使用浮点运算时出现的数值问题，但在其他很多情形中，概念的提出仅仅局限在纯粹的理论层面上。有时甚至（特别是在一些在线文章中）表述都可能不正确。在查找这类资源，评价其适用性、有效性和正确性，以及将它们改编以适应你自己的需求等方面所花的时间远远超过在其他任务上所用的时间。本书就是针对这类问题而设计编写的。本书为你提供了许多在实际应用中经常遇到的二维和三维几何学算法，并对它们进行了全面的解析和合理的编排。我们之所以将这些算法称为**几何工具算法**，是因为这些算法和概念实际上是一些用来实现应用目标的工具。

本书所涉及的不同主题的难易程度相差很大。有的非常浅显，比如计算点到线段的距离；而有的又非常复杂，比如计算两个非凸的简单闭合多面体之间的相交。有些工具算法仅需用到向量代数中的若干简单概念。而其他很多工具算法却需要使用微积分中一些较高深的概念，比如函数的导数、水平集（level sets）和使用拉格朗日乘子（Lagrange multipliers）的约束极小化（constrained minimization）。大多数书籍的重点一般都集中在某一部分章节，但我们编写的这本书却非如此。我们想使本书既适用于图形学领域的新手，也适用于经验丰富的老手。我们为需要复习向量和矩阵代数的读者提供了一章，简明地介绍了与本书相关的内容。本书提供了几个附录，其中一个附录简要地概述了三角几何的基本公式，还有一个附录涵盖了用于几何工具算法的各种不同的数值方法。

本书有两种使用方式。第一种是作为教科书使用。从某种意义上来说，本书介绍的内容是为传授算法的重要思想服务的，因此，本书适合作为大学图形学几何算法课程的教材。虽然本书没有为各章节配备练习，但本书提供了大量的伪码（pseudocode）。用一种实际的编程语言实现这些伪码就是非常合适的一组课程作业。这里引用一句著名的谚语：空谈不如实践。

本书的第二种使用方式是作为参考指南。关于算法的各章是按空间维来组织的，首先介绍二维的内容，然后再介绍三维的内容。只有关于计算几何学的那一章没有按维来组织，但它是作为独立的一章来编排的，这样也很合理。这种组织方式便于读者查找感兴趣的算法。用空间维来划分内容的尝试稍微增加了一些写作的代价。有些适用于任意维空间并且一般只需写作一次的内容现在被重复了。例如，点到线段的距离可以按不分空间维和自由

坐标的方式来描述，然而我们选择了在二维和三维中分别讨论该问题的方式。我们认为，这种选择使各章节相对独立，因而可帮助读者避免一种通常的做法，即用许多的纸片粘贴于书的不同位置以便能快速翻到与手头的问题相关的所有章节。

在计算机科学书籍中包含可运行的源代码已经成为出版界的一种惯例。在大部分情形中，编写阐明书中概念的代码需要花费相当可观的时间。对于本书这种涵盖了庞大数量的算法的大型书籍来说，提供阐明所有算法的完整代码集是很不切实际的。因为即使对商业机构来说，这也是一个艰巨的任务。作为一种替代方案，我们在书中增加了尽可能多的伪码。参考文献中包括了许多提供或连接到算法实现的 Web 网站（在本书第一次印刷时都是有效的连接）的参考条目。其中提供了大量的算法实现的一个网站是 *www.magic-software.com*，它的主机由魔术软件公司（Magic Software, Inc.）提供，Dave Eberly 负责维护。可以从这个站点免费下载该网站提供的源代码。该网站还为本书建立了一个网页，即 *www.magic-software.com/GeometricTools.html*，其中包含了与本书有关的信息、勘误表，以及代码更新历史（其中包括一些通知，说明了新增的源代码的有关信息和对老的源代码的错误更正）。网页 *www.mkp.com/gtcg* 中也提供了与本书相关的资源。

我们感谢本书的评审人员，Tomas Akenine-Möller（Chalmers University of Technology），Ian Ashdown（byHeart Consultants Limited），Eric Haines（Autodesk，Inc.），George Innis（Magic Software，Inc.），Peter Lipson（Toys for Bob, Inc.），John Stone（University of Illinois），Dan Sunday（Johns Hopkins University）和 Dennis Wenzel（True Matrix Software），以及技术编辑 Parveen Kaler（Simon Fraser University）。评审如此规模庞大且涵盖面广的书是困难的，然而他们的勤奋已经得到了回报。评审者的意见和批评已经帮助我们改进了本书许多方面的内容。我们特别感谢 Peter 和 Dennis 的意见，因为他们承担了审阅全书这一艰难的任务，并且对本书的几乎每一个方面都提出了详细的意见，既有关于细节的意见也有关于全局的意见。David M. Eberle（Walt Disney Feature Animation）提供了若干章节的许多伪码和一些额外的技术评论，我们非常感谢他的帮助。我们也感谢我们的编辑 Diane Cerra 和她的助手 Belinda Breyer，感谢她们帮助我们组编如此浩大的书籍所花费的时间，她们理解作者需要不断的鼓励才能完成如此巨量的工作，在此感谢她们所表现的耐心。本书的成功不仅是由于我们的努力，同时也是由于上述人员的共同努力。请享受阅读的乐趣吧！

目 录

第 1 章 绪论 …… 1
1.1 如何使用本书 …… 1
1.2 关于数值计算的若干问题 …… 1
1.2.1 低层问题 …… 2
1.2.2 高层问题 …… 3
1.3 各章内容概要 …… 4
第 2 章 矩阵和线性系统 …… 6
2.1 导言 …… 6
2.1.1 动机 …… 6
2.1.2 组织 …… 9
2.1.3 符号约定 …… 10
2.2 多元组 …… 10
2.2.1 定义 …… 10
2.2.2 算术运算 …… 11
2.3 矩阵 …… 11
2.3.1 符号与术语 …… 12
2.3.2 转置 …… 12
2.3.3 算术运算 …… 13
2.3.4 矩阵乘法 …… 14
2.4 线性系统 …… 17
2.4.1 线性方程 …… 17
2.4.2 两个未知数的线性系统 …… 19
2.4.3 一般线性系统 …… 20
2.4.4 减行、阶梯形和秩 …… 21
2.5 方阵 …… 22
2.5.1 对角矩阵 …… 23
2.5.2 三角形矩阵 …… 23
2.5.3 行列式 …… 24
2.5.4 逆矩阵 …… 26
2.6 线性空间 …… 28
2.6.1 数域 …… 29
2.6.2 定义和性质 …… 29
2.6.3 子空间 …… 30
2.6.4 线性组合和生成空间 …… 30

2.6.5 线性无关、维数和基底……31
2.7 线性映射……32
2.7.1 映射基础……32
2.7.2 线性映射……34
2.7.3 线性映射的矩阵表示……35
2.7.4 克莱姆定理……35
2.8 特征值和特征向量……37
2.9 欧几里得空间……38
2.9.1 内积空间……38
2.9.2 正交和标准正交集……39
2.10 最小二乘法……40
2.11 推荐的阅读材料……43
第3章 向量代数……44
3.1 向量基础……44
3.1.1 向量等价……44
3.1.2 向量加法……45
3.1.3 向量减法……46
3.1.4 向量数乘……46
3.1.5 向量加法和数乘的性质……47
3.2 向量空间……48
3.2.1 生成空间……49
3.2.2 线性无关……49
3.2.3 基底、子空间和维数……49
3.2.4 方向……51
3.2.5 基底变化……52
3.2.6 线性变换……53
3.3 仿射空间……56
3.3.1 欧几里得几何……59
3.3.2 体积、行列式和数量三重积……66
3.3.3 坐标系……67
3.4 仿射变换……69
3.4.1 仿射映射的类型……72
3.4.2 仿射映射的合成……72
3.5 重心坐标和单形……73
3.5.1 重心坐标和子空间……74
3.5.2 仿射无关……74

第 4 章　矩阵、向量代数和变换 …… 76
4.1　导言 …… 76
4.2　点和向量的矩阵表示 …… 76
4.3　加法、减法和乘法 …… 78
4.3.1　向量加法和减法 …… 79
4.3.2　点与向量的加法和减法 …… 79
4.3.3　点的减法 …… 80
4.3.4　数乘 …… 80
4.4　向量乘积 …… 80
4.4.1　点积 …… 80
4.4.2　叉积 …… 81
4.4.3　张量积 …… 84
4.4.4　正交运算符和正交点积 …… 84
4.5　仿射变换的矩阵表示 …… 88
4.6　基底变化 / 帧 / 坐标系统 …… 89
4.7　向量几何和仿射变换 …… 92
4.7.1　标记法 …… 93
4.7.2　平移 …… 93
4.7.3　旋转 …… 95
4.7.4　缩放 …… 100
4.7.5　反射 …… 104
4.7.6　剪切 …… 108
4.8　投影 …… 111
4.8.1　正射投影 …… 112
4.8.2　斜轴投影 …… 113
4.8.3　透视投影 …… 114
4.9　变换法线向量 …… 116
推荐的阅读材料 …… 118
第 5 章　二维几何图元 …… 120
5.1　线形对象 …… 120
5.1.1　隐含形式 …… 120
5.1.2　参数形式 …… 121
5.1.3　表示法之间的转换 …… 122
5.2　三角形 …… 122
5.3　矩形 …… 124
5.4　折线和多边形 …… 124
5.5　二次曲线 …… 127

5.5.1 圆 ……129
5.5.2 椭圆 ……129
5.6 多项式曲线 ……130
5.6.1 贝塞尔曲线 ……130
5.6.2 B 样条曲线 ……131
5.6.3 非均匀有理 B 样条曲线 ……131
第 6 章 二维距离 ……133
6.1 点到线形对象的距离 ……133
6.1.1 点到直线的距离 ……133
6.1.2 点到射线的距离 ……134
6.1.3 点到线段的距离 ……135
6.2 点到折线的距离 ……136
6.3 点到多边形的距离 ……138
6.3.1 点到三角形的距离 ……138
6.3.2 点到矩形的距离 ……150
6.3.3 点到正交平截面的距离 ……151
6.3.4 点到凸多边形的距离 ……154
6.4 点到二次曲线的距离 ……155
6.5 点到多项式曲线的距离 ……156
6.6 线形对象之间的距离 ……157
6.6.1 直线到直线的距离 ……157
6.6.2 直线到射线的距离 ……158
6.6.3 直线到线段的距离 ……159
6.6.4 射线到射线的距离 ……159
6.6.5 射线到线段的距离 ……162
6.6.6 线段到线段的距离 ……162
6.7 线形对象到折线或多边形的距离 ……163
6.8 线形对象到二次曲线的距离 ……164
6.9 线形对象到多项式曲线的距离 ……166
6.10 GJK 算法 ……166
6.10.1 集合运算 ……167
6.10.2 算法概述 ……168
6.10.3 其他算法 ……170
第 7 章 二维相交 ……171
7.1 线形对象之间的相交 ……171
7.2 线形对象与折线的相交 ……174
7.3 线形对象与二次曲线的相交 ……175

7.3.1 线形对象与一般二次曲线的相交 ······175
7.3.2 线形对象与圆形曲线的相交 ······176
7.4 线形对象与多项式曲线的相交 ······176
7.4.1 代数方法 ······177
7.4.2 折线逼近 ······178
7.4.3 分级包围 ······178
7.4.4 单调分解 ······179
7.4.5 栅格方法 ······180
7.5 二次曲线之间的相交 ······181
7.5.1 一般二次曲线之间的相交 ······181
7.5.2 圆形二次曲线之间的相交 ······182
7.5.3 椭圆之间的相交 ······183
7.6 多项式曲线之间的相交 ······186
7.6.1 代数方法 ······186
7.6.2 折线逼近 ······186
7.6.3 分级包围 ······186
7.6.4 栅格方法 ······187
7.7 轴分离方法 ······188
7.7.1 投影到直线上的分离 ······188
7.7.2 固定凸多边形的分离 ······189
7.7.3 运动凸多边形的分离 ······194
7.7.4 固定凸多边形的交集 ······196
7.7.5 运动凸多边形的接触点集 ······197

第 8 章 其他二维问题 ······204

8.1 三点确定的圆 ······204
8.2 与三条直线相切的圆 ······204
8.3 与圆相切于给定点的直线 ······205
8.4 通过给定点并与圆相切的直线 ······205
8.5 与两圆相切的直线 ······208
8.6 两点和给定半径决定的圆 ······213
8.7 通过一点并与一条直线相切且具有给定半径的圆 ······214
8.8 与两条直线相切且具有给定半径的圆 ······216
8.9 经过一点并与一个圆相切且具有给定半径的圆 ······218
8.10 具有给定半径并与一条直线和一个圆相切的圆 ······222
8.11 具有给定半径并与两圆相切的圆 ······225
8.12 与一条给定直线垂直并通过一个给定点的直线 ······227
8.13 位于两点之间并与该两点等距的直线 ······228
8.14 与一条给定直线平行且相距指定值的直线 ······229

8.15 与给定直线平行且垂直（水平）距离为指定值的直线……230
8.16 与给定圆相切并与给定直线垂直的直线……232

第9章 三维几何图元……234

9.1 线形对象……234
9.2 平面对象……235
9.2.1 平面……235
9.2.2 相对于一个平面的坐标系统……237
9.2.3 平面上的二维对象……238
9.3 多边形网格、多面体和有限多面体……240
9.3.1 顶点—边—面表……244
9.3.2 互连网格……244
9.3.3 复式网格……246
9.3.4 闭合网格……247
9.3.5 一致次序……247
9.3.6 柏拉图立体……249
9.4 二次曲面……253
9.4.1 三个非零特征值……253
9.4.2 两个非零特征值……254
9.4.3 一个非零特征值……256
9.5 环面……256
9.6 多项式曲线……257
9.6.1 贝塞尔曲线……258
9.6.2 B样条曲线……258
9.6.3 非均匀有理B样条曲线……259
9.7 多项式曲面……259
9.7.1 贝塞尔曲面……260
9.7.2 B样条曲面……262
9.7.3 非均匀有理B样条曲面……263

第10章 三维距离……264

10.1 导言……264
10.2 点到线形对象的距离……264
10.2.1 点到直线或射线的距离……265
10.2.2 点到折线的距离……267
10.3 点到平面对象的距离……270
10.3.1 点到平面的距离……270
10.3.2 点到三角形的距离……272
10.3.3 点到矩形的距离……277

10.3.4 点到多边形的距离 ······278
10.3.5 点到圆或圆盘的距离 ······281
10.4 点到多面体的距离 ······283
10.4.1 一般问题 ······283
10.4.2 点到有向有界箱的距离 ······285
10.4.3 点到正交平截体的距离 ······287
10.5 点到二次曲面的距离 ······291
10.5.1 点到一般二次曲面的距离 ······291
10.5.2 点到椭球面的距离 ······292
10.6 点到多项式曲线的距离 ······293
10.7 点到多项式曲面的距离 ······295
10.8 线形对象之间的距离 ······297
10.8.1 直线与直线之间的距离 ······297
10.8.2 线段 / 线段、直线 / 射线、直线 / 线段、射线 / 射线、射线 / 线段之间的距离 ··299
10.8.3 计算线段到线段的距离的另一种方法 ······310
10.9 线形对象与三角形、矩形、四面体和有向有界箱之间的距离 ······316
10.9.1 线形对象到三角形的距离 ······316
10.9.2 线形对象到矩形的距离 ······322
10.9.3 线形对象到四面体的距离 ······326
10.9.4 线形对象到有向有界箱的距离 ······328
10.10 直线到二次曲面的距离 ······341
10.11 直线到多项式曲面的距离 ······342
10.12 GJK 算法 ······343
10.13 杂项 ······343
10.13.1 直线与平面曲线之间的距离 ······343
10.13.2 直线与平面实心物体之间的距离 ······345
10.13.3 平面曲线之间的距离 ······345
10.13.4 曲面上的测地距离 ······349

第 11 章 三维相交 ······352

11.1 线形对象与平面对象的相交 ······352
11.1.1 线形对象与平面的相交 ······352
11.1.2 线形对象与三角形的相交 ······354
11.1.3 线形对象与多边形的相交 ······357
11.1.4 线形对象与圆盘的相交 ······360
11.2 线形对象与多面体的相交 ······361
11.3 线形对象与二次曲面的相交 ······365
11.3.1 线形对象与一般二次曲面的相交 ······365
11.3.2 线形对象与球面的相交 ······367

11.3.3 线形对象与椭球面的相交 ……369
11.3.4 线形对象与圆柱面的相交 ……372
11.3.5 线形对象与圆锥面的相交 ……375
11.4 线形对象与多项式曲面的相交 ……380
11.4.1 代数曲面 ……381
11.4.2 自由形态曲面 ……382
11.5 平面对象之间的相交 ……388
11.5.1 两个平面之间的相交 ……388
11.5.2 三个平面之间的相交 ……390
11.5.3 三角形与平面的相交 ……392
11.5.4 三角形与三角形的相交 ……396
11.6 平面对象与多面体的相交 ……398
11.6.1 三角网格 ……399
11.6.2 一般多面体 ……400
11.7 平面对象与二次曲面的相交 ……401
11.7.1 平面与一般二次曲面的相交 ……401
11.7.2 平面与球面的相交 ……402
11.7.3 平面与圆柱面的相交 ……404
11.7.4 平面与圆锥面的相交 ……413
11.7.5 三角形与圆锥面的相交 ……428
11.8 平面对象与多项式曲面的相交 ……431
11.8.1 埃尔米特曲线 ……432
11.8.2 几何定义 ……433
11.8.3 计算曲线 ……434
11.8.4 算法 ……435
11.8.5 实现要点 ……437
11.9 二次曲面之间的相交 ……437
11.9.1 一般相交问题 ……437
11.9.2 椭球面 ……443
11.10 多项式曲面之间的相交 ……446
11.10.1 细分方法 ……447
11.10.2 格子评测 ……447
11.10.3 解析方法 ……448
11.10.4 步进方法 ……448
11.11 轴分离方法 ……448
11.11.1 固定凸多面体的分离 ……448
11.11.2 运动凸多面体的分离 ……451
11.11.3 固定凸多面体的交集 ……453
11.11.4 固定凸多面体的接触集 ……453

11.12 杂项 ……459
11.12.1 有向有界箱与正交平截体的相交 ……459
11.12.2 线形对象与轴对齐有界箱的相交 ……461
11.12.3 线形对象与有向有界箱的相交 ……464
11.12.4 平面与轴对齐有界箱的相交 ……467
11.12.5 平面对象与有向有界箱的相交 ……468
11.12.6 轴对齐有界箱之间的相交 ……470
11.12.7 有向有界箱之间的相交 ……470
11.12.8 球面与轴对齐有界箱的相交 ……474
11.12.9 圆柱面之间的相交 ……475
11.12.10 线形对象与环面的相交 ……485

第 12 章 其他三维问题 ……488

12.1 点在平面上的投影 ……488
12.2 向量在平面上的投影 ……489
12.3 直线与平面的夹角 ……490
12.4 两平面之间的夹角 ……491
12.5 以一条直线为法线并通过一给定点的平面 ……491
12.6 三点决定的平面 ……492
12.7 两条直线之间的夹角 ……493

第 13 章 关于计算几何学的话题 ……495

13.1 二维空间分区二叉树 ……495
13.1.1 多边形的空间分区二叉树表示 ……495
13.1.2 最小分解与平衡树 ……501
13.1.3 用空间分区二叉树进行点在多边形内的检测 ……502
13.1.4 用空间分区二叉树分解线段 ……503
13.2 三维空间分区二叉树 ……505
13.2.1 多面体的空间分区二叉树表示 ……506
13.2.2 最小分解与平衡树 ……507
13.2.3 用空间分区二叉树进行点在多面体内的检测 ……508
13.2.4 用空间分区二叉树分解线段 ……509
13.2.5 用空间分区二叉树分解凸多边形 ……510
13.3 点在多边形内的检测 ……511
13.3.1 点在三角形内的检测 ……512
13.3.2 点在凸多边形内的检测 ……513
13.3.3 点在一般多边形内的检测 ……515
13.3.4 点在多边形内的快速检测法 ……520
13.3.5 栅格方法 ……520

13.4 点在多面体内的检测……521
13.4.1 点在四面体内的检测……521
13.4.2 点在凸多面体内的检测……522
13.4.3 点在一般多面体内的检测……524
13.5 与多边形有关的布尔运算……526
13.5.1 抽象运算……526
13.5.2 两种基础运算……528
13.5.3 使用空间分区二叉树的布尔运算……529
13.5.4 其他算法……532
13.6 与多面体有关的布尔运算……534
13.6.1 抽象运算……534
13.6.2 使用空间分区二叉树的布尔运算……534
13.7 凸包……536
13.7.1 二维凸包……537
13.7.2 三维凸包……548
13.7.3 高维凸包……552
13.8 德洛奈三角剖分……556
13.8.1 二维增量构建……558
13.8.2 一般维度增量构建……561
13.8.3 用凸包实现构建……564
13.9 多边形分解……565
13.9.1 一个简单多边形的可见性图……565
13.9.2 三角剖分……568
13.9.3 水平分解三角剖分……571
13.9.4 凸分解……582
13.10 外接球与内切球……589
13.10.1 外接球……590
13.10.2 内切球……591
13.11 点集的最小区域……592
13.11.1 最小面积矩形……592
13.11.2 最小体积箱体……595
13.11.3 最小面积的圆……595
13.11.4 最小体积的球……599
13.11.5 杂项……600
13.12 面积和体积测量……602
13.12.1 二维多边形的面积……602
13.12.2 三维多边形的面积……605
13.12.3 多面体的体积……608

附录 A 数值方法……610
附录 B 三角几何……682
附录 C 几何图元基础公式……700
参考文献……709
图索引……720
表索引……735

第 1 章 绪 论

1.1 如何使用本书

本书包含许多方面的内容。简单地浏览一下目录，就可以知道本书是一本关于二维和三维几何算法的书籍，这些算法可应用于计算机图形学和其他领域。本书各章节的组织方式便于读者找到感兴趣的算法，并使各章节的内容尽可能地互相独立。因此，本书非常适合作为参考书，供经验丰富的业界人士为手头的项目查找特定的算法。

然而，本书不仅仅是一本参考书。仔细地研究本书的内容就会发现，许多用于分析几何问题的概念都是通用的。例如，考虑计算不同图元对（比如点、线段、三角形、矩形、四面体或框）之间的距离这一三维几何问题。每一对图元对之间的距离都可根据有关图元形状的特定知识来分析。分析该问题的常用的统一方法是用不同的参数来表示不同的图元，即零、一、二、三分别表示点、线段、三角形或矩形、四面体或框。位于不同图元上的任意两点之间的距离平方表达式是一个包含适当参数的二次多项式。这两个图元之间的距离平方就是该二次多项式的最小值。搜索该函数的值域，将找到位于不同图元上的最近的两个点所对应的变量值，以及相应的两个图元之间的最小距离平方。搜索参数域的思想是用于计算两个凸多面体之间距离的 GJK 距离算法的基础。解决不同问题的通用思想形成了计算机图形学工作者解决新问题必须具备的一组分析工具的实现基础。因此，本书也非常适合作为学习工具，用以帮助相关人员实践计算机图形学科学。而且，我们相信，本书是一本用于计算机图形学几何算法课程的好教材。

有些读者希望，在投入几何算法分析之前，首先掌握理解这种分析所必需的基础数学工具的适当知识。为了满足他们的需要，本书的前三章简要介绍了向量和矩阵代数。本书的附录包括对三角几何公式的回顾及用于本书算法的许多数值方法的概要。我们的目的是在本书中包含足够的基础知识和先进资料，以使读者能很好地理解所有的算法。然而，实现这些算法需涉及许多的概念，为了完整地理解这些相关的概念，读者可能需要学习其他的资料。比如，有些算法需要求解多项式方程系统。可用多种方法来求解一个系统，有的方法在数值精度上更加适合于特定的算法。我们当然鼓励读者学习尽可能多的相关材料并掌握足够的知识，以解决应用中出现的各种问题。掌握的知识越多，解决实际问题的能力越强。

1.2 关于数值计算的若干问题

我们认为本书能满足许多领域的读者的需求。无论是哪个领域的读者，在计算机编程中都不可避免地面临一个难题，即必须处理浮点数系统中出现的计算问题。当然，总的来

说，在试图实现算法之前，必须先充分地理解算法的理论要点。但是仅有理论上的理解是不够的。熟悉浮点数系统的程序员都知道这一点，只有他们才能体会其中的奥妙，他们能找到许多你根本想像不到的方法，来证明你的程序逻辑是根本行不通的。

1.2.1 低层问题

用理论公式来表示几何算法时，通常采用实数。在浮点系统中编码实现这些算法时，我们将直接面临的一个问题是，并非所有的实数都表现为浮点数。如果 r 是一个实数，设 $f(r)$ 为其浮点数表示。在大多数情况下，f 被四舍五入成最接近的浮点数或者被舍去小数部分。无论采用哪种方法，用 f 表示 r 的绝对误差为 $|f(r)-r|$。其相对误差为 $|f(r)-r|/|r|$（假设 $r \neq 0$）。

浮点数的算术运算也会产生数值误差。如果 r 和 s 都是实数，它们之间的四则基本运算为：加，$r+s$；减，$r-s$；乘，$r \times s$；除，r/s。假设用 $\oplus$，$\ominus$，$\otimes$，$\oslash$ 来表示对应的浮点算术运算。则和 $r+s$ 近似表示为 $f(r) \oplus f(s)$，差 $r-s$ 近似表示为 $f(r) \ominus f(s)$，积 $r \times s$ 近似表示为 $f(r) \otimes f(s)$，商 r/s 近似表示为 $f(r) \oslash f(s)$。浮点数不会保持实数的一般算术性质。比如，如果 $s \neq 0$，则 $r+s \neq r$。但是 $f(r) \oplus f(s) = f(r)$ 是可能成立的，特别是当 $f(r)$ 在数量上比 $f(s)$ 大得多时。实数加法具有结合性和交换性。它与相加数字的次序无关。但是，浮点数加法与相加数字的次序有关。假设已有数字 r，s，以及将要与它们相加的 t。在实数运算中，$(r+s)+t = r+(s+t)$。在浮点数算术中，$(f(r) \oplus f(s)) \oplus f(t) = f(r) \oplus (f(s) \oplus f(t))$ 却不一定成立。例如，假设 $f(r)$ 在数量上比 $f(s)$ 和 $f(t)$ 大得多，则有 $f(r) \oplus f(s) = f(r)$ 和 $f(r) \oplus f(t) = f(r)$。于是有 $(f(r) \oplus f(s)) \oplus f(t) = f(r) \oplus f(t) = f(r)$。但是有可能使 $f(s) \oplus f(t)$ 足够大，使得 $f(r) \oplus (f(s) \oplus f(t)) \neq f(r)$。这样就产生了违反结合性的例子。一般地，非负浮点数求和应该按从小到大的顺序进行，以避免大数屏蔽小数。如果 r_1 到 r_n 是将要相加的数，如果它们的大小次序为 $r_{i_1} \leqslant \cdots \leqslant r_{i_n}$，则在浮点加法中的相加次序应该为 $(((f(r_{i_1}) \oplus f(r_{i_2})) \oplus f(r_{i_3})) \oplus \cdots \oplus f(r_{i_n}))$。

其他需要关注的浮点问题是：两个数量几乎相等的数相减，或者除以接近零的数所得的有效数字的消去，这两种情况都将产生不可接受的四舍五入误差。演示纠正这两种误差的方法的一个典型例子，就是求解当 $a \neq 0$ 时的二次方程 $ax^2+bx+c=0$。其理论根为

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad 和 \quad x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

假设 $b>0$ 并且 b^2 在数量上比 $4ac$ 大得多，则 $\sqrt{b^2-4ac}$ 将近似等于 b，因此 x_1 的分子由数量几乎相等的两个数之差决定，这将导致有效数字的损失。注意，x_2 并不受上述问题的影响，因为它的分子没有数字消去。一种纠正方法是，观察到 x_1 等价于

$$x_1 = \frac{-2c}{b+\sqrt{b^2-4ac}}$$

分母是两个数量相当的正数之和，因此不会出现相减产生的数字消去。但是，观察下式

$$x_2 = \frac{-2c}{b-\sqrt{b^2-4ac}}$$

在该式中，x_2 受到了相减产生的数字消去和除以接近零的数的影响。显然，仅仅选取一种求根公式并不足以适应所有的情况。为完备起见，应该考虑 b 和 $\sqrt{b^2-4ac}$ 的数量大小，并据此选用合适的公式来求解 x_1 和 x_2。

即使数值误差在合理的范围内，上述例子还说明了另一个需要处理的问题。通过分析可知，理论上 $b^2-4ac \geq 0$，因此该二次方程仅有实数根。数值的四舍五入误差将可能引起表示 b^2-4ac 的浮点数为很小的负数，这将导致平方根运算的失败（一般产生一个无效的 *NaN* [非数]）。如果理论上已知 $b^2-4ac \geq 0$，那么计算平方根的安全方法是使用公式 $\sqrt{|b^2-4ac|}$ 或 $\sqrt{\max\{0, b^2-4ac\}}$ 。

1.2.2 高层问题

进行数学思考时，浮点数系统中存在一些主要的陷阱，其中之一与排中律（Law of the Excluded Middle）有关。简单地说，一个命题非真即假。用符号术语来表述就是，如果 S 是一个布尔语句（Boolean statement）（其值非真即假），则布尔语句 S 或者非 S 总为真。实现代码时往往假定排中律总是正确的。然而在浮点算术中却并非总是如此。

【实例】假定有一个凸四边形（convex quadrilateral），按逆时针次序其顶点依次为 $V_i(0 \leq i \leq 3)$，P 为四边形内的一点，即 P 在四边形内，而不是在它的四条边上。当所有点都用实数表示时，下面的三个陈述中必有一个是正确的：

- P 在三角形 $\langle V_0, V_1, V_3\rangle$ 内。
- P 在三角形 $\langle V_1, V_2, V_3\rangle$ 内。
- P 在对角线 $\langle V_1, V_3\rangle$ 上。

在通过计算重心坐标来进行包含测试的浮点数系统中，上述所有陈述可能都是错误的！当 P 几乎位于对角线 $\langle V_1, V_3\rangle$ 上时将出现问题。此时，包含 P 的三角形的重心坐标之一在理论上是一个小正数。浮点四舍五入误差可能将该坐标变为一个小负数。这样，P 将被认为位于三角形之外。如果它也位于另一个三角形之外，则上述三个布尔条件都为假。当试图在三角形网格中判断哪个三角形包含指定点时，例如，在德洛奈三角剖分的增量构造过程中，可能会出现这种问题。■

【实例】再假定有一个凸四边形，其任意的三个顶点都构成一个三角形。其中任何一个三角形的外接圆不一定包含第四个顶点。当所有点用实数表示时，在理论上，至少有一个外接圆必将包含第四个顶点。

在浮点数系统中，浮点四舍五入误差可能导致如下的测试结果：没有任何一个外接圆包含对应的第四个顶点。当试图求解包含有限点集的最小面积圆时，可能会出现这种问题。■

【实例】理论上，凸多面体与平面的交集只能是一个点，一条线段，或者一个凸多边形。在浮点数系统中，通过计算得到的交集可能包括一个凸多边形和与其顶点相连的一条或多条线段。例如，交集可能包含四个点 $V_i(0 \leq i \leq 3)$，一个三角形 $\langle V_0, V_1, V_2\rangle$ 和一条边 $\langle V_2, V_3\rangle$。构造相交所得的多边形的程序逻辑必须处理这类异常的情况。■

几乎在与浮点数有关的任何实现中都可能出现许多像这些实例一样的问题，因此，在

构建程序逻辑时，应该时刻注意不要仅仅依赖数学推理。

在许多计算几何算法中都存在一个高层问题，即共线、共面或共点的问题。为了简化分析，关于这些算法的理论讨论往往假设这种问题不会出现。例如，在一组点的德洛奈三角剖分中，如果不存在四点共圆，则三角剖分是惟一的。构建构造三角剖分的增量算法是很简单的。然而，算法的实现必须慎重考虑出现四点共圆（或在浮点系统中出现四点近似共圆，参见低层问题中的实例）的一种或两种可能的构形。凸包的构造也受到点的共线和共面问题的困扰。

由于浮点问题的存在，我们需要仔细地实现与相交点的构造有关的特定算法。考虑计算两个椭圆的相交点的问题。正如你在稍后的章节中将读到的，该问题等价于计算具有一个变量的四次多项式的根。求根的数值算法可用于求解这类多项式方程，但是当四次项的系数为零或接近零时，应该小心求解。在这种情况下，求根过程可能误入歧途。从几何意义来说，当椭圆为圆或接近圆时，将出现这种情况。即使首项系数足够大，也可能出现另外一类问题，即偶次重根问题。如果 r 是函数 $f(x)$ 的一个奇次重根，则 $f(r)=0$，但是 f 在根的一边为负，在根的另一边为正（至少在 x 充分接近 r 时是这样）。如果 r 是偶次重根，则 f 在根的两边的符号相同（在 x 充分接近 r 时是这样）。典型的例子是，对于函数 $f(x)=x$，$r=0$ 是一个奇次重根（1）；对于函数 $f(x)=x^2$，$r=0$ 是一个偶次重根（2）。对分求根法要求根为奇次重根，因此不能用这种方法来求解函数 $f(x)=x^2$ 的根。牛顿求根法的标准形式用于求解重次为 1 的根，但我们将讨论求解更多重次根的更高级的修正形式。

求根的数值问题可简单地视为使用浮点数的一个副作用，它并不会频繁出现。然而，由于几何问题的本质特性，有时还是会出现这种情况！考虑检测两个运动椭圆第一次相交的问题。假设两个椭圆具有不同的轴长，它们第一次接触时的交集是一个单独的点。而且，此时待求根的这个四次多项式具有一个偶次重根（基于上述假设）。因此，你的求根方法必须能够处理偶次重根。在处理对象的相交问题时，奇次相重和偶次相重的概念与横截性（transversality）和相切性（tangency）相关。如果一条曲线与另一条曲线相交，并且它们各自在相交点的切线不平行，则相交是横截相交。对于这种相交的任何多项式方程都具有一个奇次重根。如果切线平行，则曲线相切接触，并且多项式方程有一个偶次重根。在许多应用中，特别是在运动物体的碰撞检测中，相切接触具有重要意义。

最后，在实现算法时考虑较少的一种现象是输入参数的次序相关性。例如，如果要实现一个检测两条线段是否相交的函数 `TestIntersection(Segment,Segment)`（返回值非 `true` 即 `false`），那么，对任意输入 `S0` 和 `S1`，我们将期望 `TestIntersection(S0,S1)` 和 `TestIntersection(S1,S0)` 产生相同的结果。如果该函数不能满足这种要求，那么可能是因为算法设计得不合理，但更可能是因为在实现时没能正确地处理浮点问题。

1.3 各章内容概要

我们为希望复习向量和矩阵代数的基本概念的读者提供了第 2 章，第 3 章和第 4 章。附录 A 提供了用于本书算法的许多数值方法的概要。三角几何公式可在附录 B 中找到。附录 C 是一个关于在本书中将遇到的一些几何图元基本公式的快速参考。

第 5 章提供了几何问题涉及的各种二维对象的定义，包括直线、射线、线段、多边形、二次曲线段（二次方程定义的曲线），以及多项式曲线。主要的几何问题包括在第 6 章讨论的几何测量和在第 7 章讨论的相交问题。第 8 章提供了一些重要的杂项问题。

第 9 章提供了几何问题涉及的各种三维对象的定义，包括直线、射线、线段、平面和面形对象（内嵌于三维平面中的二维对象）、多面体和多边形网格、二次曲面（二次方程定义的曲面）、多项式曲线、多项式曲面、有理曲线，以及有理曲面。主要的几何问题包括在第 10 章讨论的几何测量和在第 11 章讨论的相交问题。第 12 章提供了一些重要的杂项问题。

在第 13 章中提供了关于计算几何学主题的大量材料。这些主题包括空间分区二叉树、对多边形和多面体的布尔运算、点在多边形内和点在多面体内的测试、点集所组成凸包的构造、点集的德洛奈三角剖分、多边形的三角剖分和多边形的凸块分解，以及点集定义的有边界容器的最小面积和体积。本书还包含一节内容，讨论了二维或三维中多边形面积的计算和多面体体积的计算。

第 2 章　矩阵和线性系统

2.1　导言

本书的目标之一是提供大量的解决计算机图形学中许多常见问题的“处方”。虽然我们的意图是解释这些处方是如何工作的，但我们还将走得更远些。“授之以鱼，不如授之以渔”，这是一句古老的谚语。为了实现这一目标，我们在本书中提供了几章内容，试图帮助读者理解计算机图形学中的基本几何工具算法是如何工作的，以及为什么会这样工作。当你遇到一个新问题时，只要本书讨论过相同类型的问题，你就能得出解决问题的方法，不但能直接采用我们提供的处方，而且能真正理解构建我们的处方的概念、原理和技术。

2.1.1　动机

大多数与计算机图形学相关的书籍都包含一章或一个附录用以说明点、向量和矩阵运算的基本背景。在这方面，本书也遵循这种惯例。但我们的做法不太一样。许多计算机图形学书籍都包括用于计算机图形学的数学分析，它们都首先论述基于坐标的面向矩阵方法。许多商业或研究中使用的图形库的接口也普遍采用这种方法。

基于坐标的方法强调基于与特定坐标系统相关的几何实体之间关系的分析。该方法在一些情形中很有用，例如，设想有一个层次定义模型，要得到定义在层次的不同部分上的两点之间的距离，就必须将其中一个点的坐标转换到另一点所在的空间，并用常见的形式来确定欧几里得距离。

然而，即使是在简单的例子中，也能发现这种方法的一些缺点。考虑 DeRose（1989）给出的一个例子，它简单地显示基于矩阵的变换代码。在缺少上下文信息时，其中计算的真实意思显得含混不清。考虑变换一个二维点的几行类 C 代码：

```
float P[2];
float PPrime[2];
float M[2][2];

P[0] = x;
P[1] = y;

M[0][0] = 3; M[0][1] = 0;
M[1][0] = 0; M[1][1] = 2;

PPrime[0] = P[0] * M[0][0] + P[1] * M[1][0];
PPrime[1] = P[0] * M[0][1] + P[1] * M[1][1];
```

上述代码段可按如下三种方式中的任何一种来解释：

（1）改变坐标，在几何上不改变点，只是改变坐标系统（参见图 2.1（a））。

（2）坐标平面内的变换，移动点，但不改变坐标系统（参见图 2.1（b））。

（3）从一个平面到另一个平面的变换（参见图 2.1（c））。

正如 DeRose 所指出的，这些解释是不能相互转化的：第一种解释不会改变长度和角度，但第二种和第三种解释会改变长度和角度。

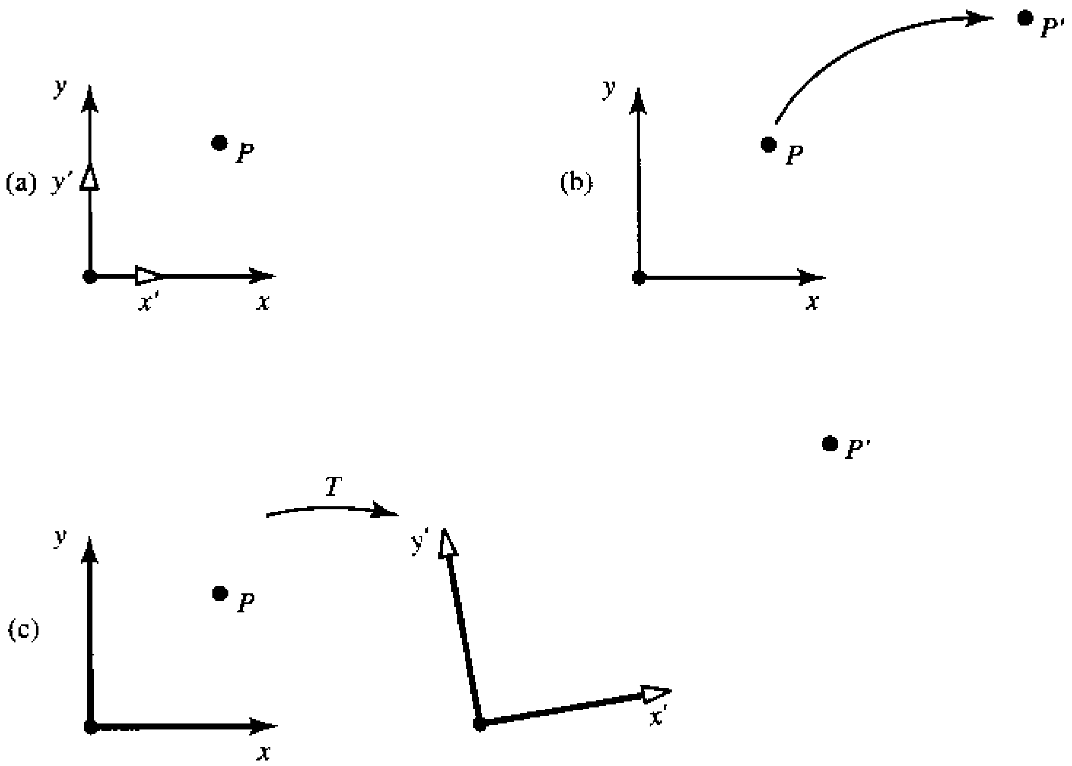


图 2.1　对示例的不明确变换的多种不同解释：（a）改变坐标；（b）平面内的变换；（c）从一个平面到另一个平面的变换

从上述例子中可以发现一个更加复杂的问题，即没有任何线索说明P表示点还是表示向量。因此，以这种方式编写和概念化代码会执行 Ron Goldman（1985）所说的“非法”操作，比如将两个点相加在一起。

这样的参数在技术上很可能被认为是正确的，但是一旦执行了这样的代码，会造成怎样的损害呢？事实表明，过分依赖严格基于坐标的方法不但会导致不明确的实现代码，并且会隐含非法运算的可能性，而且会误导人们用梦魇般的代码来实现一个在概念上相对简单的问题。下面是 Miller（1999a，1999b）提供的一个极好的例子。假定有向量$\vec{u}$和$\vec{v}$，求将$\vec{u}$旋转到$\vec{v}$上的变换矩阵（注意，将一个向量旋转到另一个向量上有无数种可能情形；我们在此仅考虑位于一个平面上、具有最小夹角的两个向量的旋转）。如果使用严格基于坐标的方法，求解步骤如下：

第 1 步　确定变换的列，比如z轴，每个向量都将映射到该列。

第 2 步　连接对$\vec{u}$的变换和对$\vec{v}$的变换的逆。

和 Miller 的建议一样，我们仅观察其中的一小部分计算序列，例如，用下面的方法（Foley 等 1996）将$\vec{u}$变换到z轴上，我们得到的矩阵是如下的乘积

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{\sqrt{u_x^2+u_z^2}}{\|\vec{u}\|} & \frac{-u_y}{\|\vec{u}\|} \\ 0 & \frac{u_y}{\|\vec{u}\|} & \frac{\sqrt{u_x^2+u_z^2}}{\|\vec{u}\|} \end{bmatrix} \begin{bmatrix} \frac{u_x}{\sqrt{u_x^2+u_z^2}} & 0 & \frac{u_z}{\sqrt{u_x^2+u_z^2}} \\ 0 & 1 & 1 \\ \frac{-u_z}{\sqrt{u_x^2+u_z^2}} & 0 & \frac{u_x}{\sqrt{u_x^2+u_z^2}} \end{bmatrix}$$

Miller 指出了使用这种方法的几个难点（其中还不包括上述推导中明显的丑陋之处）：式中的分母，特别是第二个矩阵中的分母可能为零或接近零。这就要求实现代码在使用这些值之前仔细地对它们进行范围检查。如果$\vec{u}$碰巧平行于（或几乎平行于）y轴，就会出现这种情形。然而从几何意义上来说，该条件与我们讨论的问题是毫不相关的。这类问题也存在于类似的向量$\vec{v}$的变换矩阵中。最后还必须反转向量$\vec{v}$的两个矩阵，这将可能引起数值的不精确。

注意到上述问题可被视为计算向量$\vec{w} = \vec{u} \times \vec{v}$的旋转变换矩阵**M**，可以得到解决上述问题的一种替代方法，即基于向量的方法。首先计算$\vec{u}$和$\vec{v}$之间夹角θ的正弦和余弦：

$$\sin\theta = \frac{\|\vec{u} \times \vec{v}\|}{\|\vec{u}\|\|\vec{v}\|}$$

$$\cos\theta = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|}$$

据 Goldman（1990b）研究，这样的变换可用如下的公式来计算：

$$\mathbf{M} = \cos\theta\mathbf{I} + (1 - \cos\theta)\hat{w} \otimes \hat{w} + \sin\theta\mathbf{W}$$

其中

$$\hat{w} = \frac{\vec{w}}{\|\vec{w}\|}$$

$$\mathbf{W} = \begin{bmatrix} 0 & -w_z & w_y \\ w_z & 0 & -w_x \\ -w_y & w_x & 0 \end{bmatrix}$$

I为单位矩阵，$\otimes$为张量积运算符（参见 4.4.3 节）。

与基于坐标的方法相辉映的是向量几何方法，上述就是使用该方法的一个例子。这种替代方法强调将点和向量作为抽象的几何实体来考虑，并用几何运算符（比如变换、叉积等）来操作（也是抽象的）它们，而不是从一开始就考虑点元（或向量），也不是直接将变换看成 3×3 或 4×4 矩阵。简单地说，我们说的就是几何代数（geometric algebra）。DeRose（1989）和 Goldman（1987）将这种方法称为与坐标无关的几何学（coordinate-free geometry），以强调这种方法与常用的基于坐标的方法之间的差别。

下一章的许多节内容将试图建立起这种几何代数的基础。这样做的动机有几个。首先，读者只有牢固地领会了算法的实质（比如，叉积到底有什么用？它的公式为什么会是这样的？），才能很好地理解本书所呈现的算法。其次，对这些基本原理的深入理解，将使读者能自己构造正确且健壮的算法来解决遇到的问题。最后，作者认为，这种方法更加直观，将其用于图形学领域可以帮助克服一般方法的弊端，如不分青红皂白地将点和向量表示成实数数组，然后用不同的运算符（点积、叉积、变换）显性地操作这些数组成员的任意算术组合。

总之，本书将尽可能地采用向量几何学方法来讨论和解释问题，相应地，我们将尽可能清晰地区分点和向量。支持这种区分的方法是广泛用于计算机图形学文献的符号标记惯例：

- 点在方程中表示为大写的 Times 字体斜体字符，通常使用字母 P，Q，R等；而在需要表示次序或点集时，使用下标：P_1，P_2等。
- 向量表示为小写的 Times 字体斜体字符，上面带一个用来作为区别标记的箭头，通常使用字母$\vec{u}$，$\vec{v}$，$\vec{w}$。而在需要表示它们的集合、次序或数组时，使用下标：$\vec{v}_1$，$\vec{v}_2$等。单位长度向量用一个“帽子”而不是区别箭头来表示：$\hat{u}$，$\hat{v}$，$\hat{w}$。

Ron Goldman（1985）证实了这种符号能达到最大的视觉区分并能最好地反映方法原理：

在向量几何中的坐标方法中，点和向量都表示为 3 个直角坐标，这是产生许多混乱的“祸首”。如果固定坐标系的原点，则在点与向量之间存在一个自然的一一对应。因此，点 P 可用向量OP 来表示，其中O为坐标系原点。但是它们之间的细微差别往往被忽略。即使如此，等式$P = OP$也是不正确的。大象不是香蕉，点也不是向量。

2.1.2　组织

大部分讲述用于计算机图形学的几何学的书籍一般将点、向量、变换和矩阵等内容统统混在一起来讨论，但我们将采用一种不同的方法。

尽管我们在上一节中对基于坐标的方法提了一些意见，但理解矩阵和线性代数还是必不可少的。理由之一是，仿射空间（我们将在下章讨论有关内容）是一个向量空间，它与线性系统密切相关。理由之二是矩阵运算能够（也通常）用于实现向量几何运算。因此，本章展示了与后续的向量代数内容相关的矩阵和线性代数原理，以及实现向量代数的矩阵用法。深谙线性代数的读者可以直接跳到下一章。我们之所以在本书的正文中包括这部分材料，是为了满足那些喜欢“完整内容”的读者，并提供一个更好的连接矩阵、线性代数和向量代数的叙述流程。

第 3 章将从一种与坐标完全无关的方法出发，完整地论述向量代数。当然，读者将发现，许多的材料直接与本章介绍的基于线性代数的内容“重叠”。例如，本章基于抽象的线性代数的观点论述了向量空间，而下一章将基于更加具体、形象的有向线段的观点来解释向量空间。当然，实际上，它们都是相同的向量空间。

第 4 章明确地将向量代数、线性代数和矩阵集成在一起。其他处理它们之间的关系的方法不是简单地将它们混在一起（这将模糊基于向量代数的本质概念），就是认为向量代数“仅仅”是线性代数的几何解释。我们的观点是：位置、方向、距离和角度等是更基本的概念，而线性代数和矩阵只是表示和操作它们的一种方法而已。这种差别有点像本质上没有解决的“宗教”或哲学问题，但是，显而易见的是，在任何情形中，与坐标无关的向量代数方法更加有利于培养直觉知识。例如，如果从点积的线性代数定义出发，那将很难理解，为什么数组元素算术运算的任意序列与向量之间的夹角有关系？到底有什么关系？但是，如果你理解了点积的几何定义并知道如何用矩阵运算来实现点积，那么你将理解点积的真正意义，并可能在试图解决新的几何问题时使用它。

2.1.3 符号约定

本书包含大量的方程、图表、代码和伪码。为了提高可读性，我们采用了一组一致的符号标记约定。表 2.1 说明了该约定的要点。

表 2.1 本书使用的数学符号

实 体	数学符号表示	伪 代 码
集合	$\{a, b, c\}$	
标量	$\alpha, \beta, \gamma, a, b, c$	`float alpha, a;`
角度	θ, ϕ	`float theta, phi;`
点	P, Q, R, P_1, P_2	`Point2D p, q, r1;Point3D p1, p2;`
向量	$\vec{u}, \vec{v}, \vec{w}$	`Vector2D u, v; Vector3D w;`
单位向量	$\hat{u}, \hat{v}, \hat{w}$	`Vector2D uHat, vHat; Vector3D wHat;`
垂直向量	$\vec{u}_\perp, \vec{v}_\perp$	`Vector2D uPerp, vPerp;`
平行向量	$\vec{u}_\parallel, \vec{v}_\parallel$	`Vector2D uPar, vPar;`
向量长度	$\lVert\vec{u}\rVert$	
矩阵	$\mathbf{M}, \mathbf{N}, \mathbf{M}_1, \mathbf{M}_2$	`Matrix3x3 m, m; Matrix4x4 m1, m2;`
转置矩阵	$\mathbf{M}^\mathrm{T}, \mathbf{N}^\mathrm{T}$	`Matrix3x3 mTrans, nTrans;`
逆矩阵	$\mathbf{M}^{-1}, \mathbf{N}^{-1}$	`Matrix3x3 mInv, nInv;`
多元组	$\mathbf{a} = (a_1, a_2, \ldots, a_n)$	
行列式	$\lvert\mathbf{M}\rvert$ or $\det(\mathbf{M})$	`Det(m)`
空间（线性等）	$\mathcal{V}, S^2$	
空间（实）	$\mathbb{R}, \mathbb{R}^2, \mathbb{R}^3$	
点积	$a = \vec{u} \cdot \vec{v}$	`a = Dot(u, v);`
叉积	$\vec{w} = \vec{u} \times \vec{v}$	`w = Cross(u, v);`
（外）张量积	$\vec{w} = \vec{u} \otimes \vec{v}$	`w = Outer(u, v);`

2.2 多元组

在进入关于矩阵的讨论之前，我们先讨论一个没有那么抽象的概念，即多元组。多元组本质上就是一个元素的有序序列。由于本书讲述的是应用于计算机图形学的几何学，因此我们将讨论限制在实数范围内。然而，应该记住，多元组和矩阵（在概念上）可以由复数或任意类型构成，而且我们讨论的许多内容（特别是关于性质的部分）都适合于任意元素类型。

2.2.1 定义

单独的一个实数通常称为一个数量，例如，6.5，4.2 或π。如果有两个数量，并且按照有意义的次序将它们组合在一起，我们就把它们叫做有序对（ordered pair）；三个一组的叫

做有序三元组（ordered triple）；四个一组的叫做有序四元组（ordered quadruple）；依此类推。这种序列的通用术语是多元组（tuple）。我们暂且用 Roman 字体加粗来标记它们，并用圆括弧将它们的元素括起来。例如：

$$\mathbf{a} = (6.5, 42)$$

$$\mathbf{b} = (\pi, 3.75, 8, 15)$$

一般地，我们将含有 n 个元素的多元组称为 n 维多元组，并用下标来标记：$\mathbf{a} = (a_1, a_2, \ldots, a_n)$。

2.2.2 算术运算

我们感兴趣的是由实数组成的多元组，当然我们也需要了解使用多元组的算术运算。

如果每个多元组具有相同的元素个数，并且元素都表示相应的数量，则多元组的加法（或减法）有意义。此时，两个多元组的相加（或相减）可以简单地通过将它们对应的元素相加（相减）来实现，如下所示：

$$\mathbf{a} = (a_1, a_2, \cdots, a_n)$$

$$\mathbf{b} = (b_1, b_2, \cdots, b_n)$$

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1, a_2 + b_2, \cdots, a_n + b_n)$$

$$\mathbf{a} - \mathbf{b} = (a_1 - b_1, a_2 - b_2, \cdots, a_n - b_n)$$

例如，(6，3，7) + (1，−2，4) = (7，1，11)。

多元组与数量的乘法和除法定义为，对多元组的每一个元素简单地实施乘法（除法）：

$$k\mathbf{a} = (ka_1, ka_2, \cdots, ka_n)$$

$$\frac{\mathbf{a}}{k} = \left(\frac{a_1}{k}, \frac{a_2}{k}, \cdots, \frac{a_n}{k}\right)$$

例如，2 × (6，3，7) = (12，6，14)，以及(6，3，7) / 2 = (3，1.5，3.5)。

如何进行两个多元组的乘法呢？这是一个显而易见的问题，但是它的答案并不像加 / 减法和数量乘 / 除法那么直接。事实上，有两种类型的多元组与多元组之间的乘法，但在介绍足够多的相关知识之前，我们暂时不讨论它们。

有了多元组的概念，我们将很自然地考虑多元组的集合，它们将具有特定的意义和功能（不管是该术语的一般定义还是特定含义）。这种令我们感兴趣的多元组的组织就是矩阵，本章的其余内容将集中地讨论矩阵的表示、性质和应用。

2.3 矩阵

从其本质上来说，矩阵只不过是项目的矩形排列，其中的元素是实数或表示实数的符号。在计算机图形学的书籍中，经常在一个非常“机械”的层次上来论述矩阵，即矩阵被视为装数据的“袋子”及数据的运算规则，其中的数据用于表示和操作图形对象。然而，这种论述不能说明矩阵工作的原理。因此，在接下来的几章中，我们将试图用一种直觉的

方式把线性代数、矩阵和计算机图形几何学集成在一起。为了实现这一目标，我们鼓励你尽量将矩阵看成多元组的序列或者“多元组的多元组”，其中的次序不仅是“它工作的方式”，而且具有更深的含义。

基于不同的理由，多元组的序列（或多元组）可被方便地表示成由各个水平方向的多元组构成的从上到下的多元组堆，或者由各个垂直方向的多元组构成的从左到右的多元组群。

矩阵的传统标记法是用特定的分隔符在矩阵的左、右方把矩阵括起来。在本书中，我们使用方括号，例如：

$$[3.2 \quad 7] \qquad \begin{bmatrix} a \\ b \\ c \end{bmatrix} \qquad \begin{bmatrix} 4 & 7 & 93.5 \\ 5 & 9 & 12 \end{bmatrix}$$

2.3.1　符号与术语

我们用粗体大写字母来表示矩阵，比如，**M**或**A**。矩阵中的每一项叫做元素（element）。元素的水平或垂直排列（即多元组）分别叫做行（row）和列（column）。矩阵的行数和列数一般分别用m和n来表示，相应地，用“m 行n 列”来说明矩阵的大小，表示为$m \times$ n。如果 $m = n$，则这样的矩阵叫做方阵（square）。

如果要一般地引用矩阵的元素，我们将使用如下的常用惯例表示法：

$$\mathbf{M} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

注意，下标按照（行，列）的次序排列。

如果要更一般地引用矩阵，我们将使用如下的表示法：$\mathbf{A} = [a_{i,j}]$，表示矩阵**A**的元素如上例所示。

2.3.2　转置

一个$m \times$ n的矩阵**M**的转置矩阵（transpose）可用如下方法来生成：将**M**的m行（按次序）作为新矩阵的列（当然，**M**的列便作为新矩阵的行）。也可以认为转置矩阵的的生成方式如下：沿着矩阵左上角到右下角的对角线旋转矩阵。得到的矩阵**M**的转置矩阵表示为$\mathbf{M}^{\mathrm{T}}$，其大小自然是$n \times m$。下面我们来转置最初的例子中的矩阵：

$$\mathbf{M}_1 = [3.2 \quad 7] \qquad \mathbf{M}_1^{\mathrm{T}} = \begin{bmatrix} 3.2 \\ 7 \end{bmatrix}$$

$$\mathbf{M}_2 = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \qquad \mathbf{M}_2^{\mathrm{T}} = [a \quad b \quad c]$$

$$\mathbf{M}_3 = \begin{bmatrix} 4 & 5 & 93.5 \\ 5 & 9 & 12 \end{bmatrix} \qquad \mathbf{M}_3^{\mathrm{T}} = \begin{bmatrix} 4 & 5 \\ 5 & 9 \\ 93.5 & 12 \end{bmatrix}$$

一般地，如果有矩阵

$$\mathbf{M}=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

则其转置矩阵为：

$$\mathbf{M}^{\mathrm{T}}=\begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{m,n} \end{bmatrix}$$

矩阵的转置具有几个值得提及的性质：

i．$(\mathbf{AB})^{\mathrm{T}}=\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$

ii．$(\mathbf{A}^{\mathrm{T}})^{\mathrm{T}}=\mathbf{A}^{\mathrm{T}}$

iii．$(\mathbf{A}+\mathbf{B})^{\mathrm{T}}=\mathbf{A}^{\mathrm{T}}+\mathbf{B}^{\mathrm{T}}$

iv．$(k\mathbf{A})^{\mathrm{T}}=k(\mathbf{A}^{\mathrm{T}})$

2.3.3 算术运算

在规则上，矩阵的加法和减法，以及矩阵与数量的乘法和除法，与多元组的同种运算相同。而且，它们的运算性质（交换性，结合性等）也相同。

1．加法和减法

两个矩阵的加法是多元组加法的自然扩展：如果有两个矩阵$\mathbf{A}=[a_{i,j}]$和$\mathbf{B}=[b_{i,j}]$，则它们之和可以通过对每一个多元组（行）求和而得到：

$$\mathbf{A}+\mathbf{B}=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}+\begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,n} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m,1} & b_{m,2} & \cdots & b_{m,n} \end{bmatrix}$$

$$=\begin{bmatrix} a_{1,1}+b_{1,1} & a_{1,2}+b_{1,2} & \cdots & a_{1,n}+b_{1,n} \\ a_{2,1}+b_{2,1} & a_{2,2}+b_{2,2} & \cdots & a_{2,n}+b_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}+b_{m,1} & a_{m,2}+b_{m,2} & \cdots & a_{m,n}+b_{m,n} \end{bmatrix}$$

2．数量乘法和除法

矩阵与数量的乘法的定义类似于多元组与数量的乘法：用数量与每个元素相乘。因此，如果有一个数量k和矩阵$\mathbf{A}$，则$k\mathbf{A}$定义为

$$k\mathbf{A}=\begin{bmatrix} ka_{1,1} & ka_{1,2} & \cdots & ka_{1,n} \\ ka_{2,1} & ka_{2,2} & \cdots & ka_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ ka_{m,1} & ka_{m,2} & \cdots & ka_{m,n} \end{bmatrix}$$

矩阵与数量的除法与此类似。

3．零矩阵

我们在前面曾经提到，矩阵的加法展现了普通（数量）加法的许多性质。其中一个性质是，存在一个加法单位元素（additive identity element），即存在一个叫做0矩阵的$m \times n$的矩阵，对任意矩阵M都具有如下性质：$\mathbf{M} + \mathbf{0} = \mathbf{M}$。零矩阵还具有如下性质：$\mathbf{M0} = \mathbf{0}$，正如数0在数量乘法中的表现一样。一个$m \times n$的零矩阵的所有元素都为0，有时记为$\mathbf{0}_{m \times n}$：

$$\mathbf{0}_{2\times3} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{0}_{3\times2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}$$

4．算术运算的性质

由于我们根据多元组的运算来定义矩阵的上述算术运算，而多元组的运算是根据其数量元素的算术运算来定义的，因此，矩阵运算的性质与数量运算的性质相同，这应该在我们的意料之中：

i．加法的交换性：$\mathbf{A} + \mathbf{B} = \mathbf{B} + \mathbf{A}$

ii．加法的结合性：$\mathbf{A} + (\mathbf{B} + \mathbf{C}) = (\mathbf{A} + \mathbf{B}) + \mathbf{C}$

iii．数量乘法的结合性：$k\,(l\mathbf{A}) = (kl)\,\mathbf{A}$

iv．数量乘法对加法的分配性：$k\,(\mathbf{A} + \mathbf{B}) = k\mathbf{A} + k\mathbf{B}$

v．数量加法对乘法的分配性：$(k_1 + k_2)\,\mathbf{A} = k_1\mathbf{A} + k_2\mathbf{A}$

vi．加法逆元：$\mathbf{A} + (-\mathbf{A}) = \mathbf{0}$

vii．加法单位元：$\mathbf{A} + \mathbf{0} = \mathbf{A}$

viii．数量乘法单位元：$1 \cdot \mathbf{A} = \mathbf{A}$

ix．零元素：$0 \cdot \mathbf{A} = \mathbf{0}$

我们将乘法单位元和乘法逆元留到下一节再介绍。

2.3.4　矩阵乘法

我们知道，矩阵加法是数量加法的扩展，但是，与此不同，矩阵乘法并非数量乘法的简单扩展。

1．多元组乘法

正如可以根据多元组的加法来定义矩阵的加法，我们也可以根据多元组的乘法来定义矩阵的乘法。但是什么是多元组的乘法？让我们从现实世界中的例子说起。假定有一个多元组$\mathbf{a} = (2, 3, 2)$，它列出了三个不同项目的体积（比如，砾、沙和水泥，混凝土的成分）；另一个多元组$\mathbf{b} = (20, 15, 10)$，它列出了各成分单位体积的重量。总重量是多少？显然，只需将体积和单位体积重量成对相乘，并把所得的乘积相加，即可求得结果：

$$\mathbf{ab} = (2 \times 20) + (3 \times 15) + (2 \times 10) = 105$$

这就是数量积（scalar product），由于它通常记为$\mathbf{a} \cdot \mathbf{b}$，因此又叫做点积（dot product）。一般地，如果有两个n维多元组$\mathbf{a} = (a_1, a_2, \cdots, a_n)$和$\mathbf{b} = (b_1, b_2, \cdots, b_n)$，则它们的乘积计算如下：

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_1 b_2 + \cdots + a_n b_n$$

2. 多元组乘法的性质

由于多元组的乘法就是根据数量加法和乘法来定义的，因此多元组乘法符合数量乘法的规则，这也并不令人吃惊。

i．交换性：$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$

ii．结合性：$(k\mathbf{a}) \cdot \mathbf{b} = k(\mathbf{a} \cdot \mathbf{b})$

iii．分配性：$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \cdot \mathbf{b}) + (\mathbf{a} \cdot \mathbf{c})$

3. 矩阵与矩阵相乘

正如我们将在本书的后续部分中将看到的，矩阵乘以矩阵的运算是矩阵最重要的用法之一。与你可能的猜想一样，矩阵乘法是多元组乘法的扩展，正如矩阵加法和矩阵的数量乘法是多元组加法和多元组数量乘法的扩展一样。

然而，矩阵乘法具有一个重要的特征，该特征显得并不直观或者并不明显。我们从仅仅包含一个n维多元组的矩阵的乘法开始讨论。仅有一个n维多元组的矩阵可以写成一个$n \times 1$矩阵（仅有一列），也可写成一个$1 \times n$矩阵（仅有一行）。如果有两个矩阵$\mathbf{A}$和$\mathbf{B}$，它们分别仅仅包含一个n维多元组$\mathbf{a}$和$\mathbf{b}$，那么如果$\mathbf{A}$写成行矩阵，$\mathbf{B}$写成列矩阵，则它们可以相乘，并且相乘的次序如下：

$$\mathbf{AB} = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n \tag{2.1}$$

从中可以看出，一行与一列相乘得到一个单一的实数。因此，如果有两个具有多行和多列的矩阵，那么它们的乘积自然应该包含多个实数。

一般矩阵乘法的运算法则定义如下：一个$m \times n$矩阵$\mathbf{A}$和一个$n \times r$矩阵$\mathbf{B}$，它们的乘积$\mathbf{AB}$是一个大小为$m \times r$的矩阵$\mathbf{C}$，其元素$c_{i,j}$为$\mathbf{A}$的第i行和$\mathbf{B}$的第j列的点积：

$$\mathbf{AB} = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,r} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,r} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n,1} & b_{n,2} & \cdots & b_{n,r} \end{bmatrix}$$

$$= \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + \cdots + a_{1,n}b_{n,1} & a_{1,1}b_{1,2} + a_{1,2}b_{2,2} + \cdots + a_{1,n}b_{n,2} & \cdots & a_{1,1}b_{1,r} + a_{1,2}b_{2,r} + \cdots + a_{1,n}b_{n,r} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + \cdots + a_{2,n}b_{n,1} & a_{2,1}b_{1,2} + a_{2,2}b_{2,2} + \cdots + a_{2,n}b_{n,2} & \cdots & a_{2,1}b_{1,r} + a_{2,2}b_{2,r} + \cdots + a_{2,n}b_{n,r} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1}b_{1,1} + a_{m,2}b_{2,1} + \cdots + a_{m,n}b_{n,1} & a_{m,1}b_{1,2} + a_{m,2}b_{2,2} + \cdots + a_{m,n}b_{n,2} & \cdots & a_{m,1}b_{1,r} + a_{m,2}b_{2,r} + \cdots + a_{m,n}b_{n,r} \end{bmatrix}$$

例如，如果

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 9 & 1 \end{bmatrix}$$

并且

$$\mathbf{B} = \begin{bmatrix} 1 & 7 & 5 \\ 4 & 6 & 8 \end{bmatrix}$$

那么

$$
\begin{aligned}
C &= AB \\
&= \begin{bmatrix} 2 & 3 \\ 9 & 1 \end{bmatrix} \begin{bmatrix} 1 & 7 & 5 \\ 4 & 6 & 8 \end{bmatrix} \\
&= \begin{bmatrix} 2\times1+3\times4 & 2\times7+3\times6 & 2\times5+3\times8 \\ 9\times1+1\times4 & 9\times7+1\times6 & 9\times5+1\times8 \end{bmatrix} \\
&= \begin{bmatrix} 14 & 32 & 34 \\ 13 & 69 & 53 \end{bmatrix}
\end{aligned}
$$

4. 矩阵乘法的性质

与矩阵的数量乘法和加法不同，矩阵乘法并不具有实数乘法的所有性质。

- 交换性：矩阵乘法不具备该性质。如果要让矩阵A乘以矩阵B，则A的列数必须等于B的行数，但是A的行数和B的列数可以不相等。但是这样一来，如果要将B乘以A，由于A的行数和B的列数可能不相等，因此乘法可能失败。即使A的行数和B的列数相等，结果也可能不相同。
- 结合性：如果有A (BC)，则(AB) C有定义而且等于A (BC)。
- 数量乘法的结合性：如果AB是合法的运算，则$(kA)\,B = k\,(AB)$。
- 乘法对加法的分配性：如果A是$m \times n$矩阵，B和C是$n \times r$矩阵，则$A\,(B + C) = AB + AC$。注意，由于交换性不成立，因此$(B + C)\,A = BA + CA$的结果与前式计算的结果不同。

5. 行或列多元组与一般矩阵相乘

在计算机图形学中，两种最常用的与矩阵有关的运算是两个方阵的乘法（正如我们在上一节中描述的那样）和行矩阵或列矩阵与方阵的乘法。

我们已经定义了多元组的乘法和计算元素 $c_{i,j}$ 的规则。如果将它们合在一起，我们将发现多元组之间乘法的矩阵表示必须依照表达式（2.1）所示的次序，即行多元组必须在左边，而列多元组在右边。考虑一对两个元素的多元组及它们之积：

$$
\begin{aligned}
\mathbf{a} &= (a_1, a_2) \\
\mathbf{b} &= (b_1, b_2) \\
\mathbf{ab} &= a_1b_1 + a_2b_2
\end{aligned}
$$

如果一定要用列矩阵乘以行矩阵，则我们将得到的是

$$
\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} [\, b_1 \quad b_2 \,] = \begin{bmatrix} a_1b_1 & a_1b_2 \\ a_2b_1 & a_2b_2 \end{bmatrix}
$$

如果我们采用矩阵乘法的规则，那么得到的结果会与多元组乘法有冲突。

这一结果可以扩展到A或B之一为一般矩阵的情形。得出的结论是，一个多元组与矩阵的乘法必须使这个多元组或者作为行矩阵位于左边，或者作为列矩阵位于右边。但是，我们不能简单地改变这种乘法的次序，因为矩阵乘法不具有交换性。

矩阵转置的第一个性质（参见 2.3.2 节）告诉我们，矩阵乘积的转置等于每个矩阵的转置之积，只是相乘的次序相反：$(AB)^T = B^TA^T$。这说明，如下的两个关于多元组 **a** 与一

般矩阵 **B** 相乘的表达式是相等的：

$$[a \quad b]\begin{bmatrix} c & d \\ e & f \end{bmatrix} = [ac+be \quad ad+bf]$$

$$\begin{bmatrix} c & e \\ d & f \end{bmatrix}\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ca+eb \\ da+fb \end{bmatrix}$$

稍后我们将看到，多元组的一个功能可以用多元组与矩阵的乘法来方便地实现。上述讨论清楚地表明，多元组可表示为行矩阵或列矩阵，并可相应地使用该矩阵或其转置矩阵。两种表示方法都可见于计算机图形学文献中，本书也同时使用这两种表示方法。在阅读书籍或文章时，请留意作者所使用的是哪种表示方法。所幸的是，这两种方法之间的转换是非常简单的：只需反转矩阵和向量的次序，并使用它们的转置矩阵。例如：

$$\vec{u}\mathbf{M} = [u_1 \quad u_2 \quad u_3]\begin{bmatrix} m_{1,1} & m_{1,2} & m_{1,3} \\ m_{2,1} & m_{2,2} & m_{2,3} \\ m_{3,1} & m_{3,2} & m_{3,3} \end{bmatrix}$$

$$\equiv \begin{bmatrix} m_{1,1} & m_{2,1} & m_{3,1} \\ m_{1,2} & m_{2,2} & m_{3,2} \\ m_{1,3} & m_{2,3} & m_{3,3} \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \mathbf{M}^{\mathrm{T}}\vec{u}^{\mathrm{T}}$$

2.4 线性系统

线性系统是线性代数的重要组成部分，因为线性代数中许多重要的问题都可以通过线性系统的运算来解决。可以说线性系统是非常抽象的，它可以用实数方程、复数方程或任意数据域方程来表示。但是，基于本书的目的，我们将讨论局限于实数域ℝ。

2.4.1 线性方程

线性方程的各项不是线性项（实数与次数为 1 的变量之积）就是常数（只是一个实数）。例如：

$$5x + 3 = 7$$

$$2x_1 + 4 = 12 + 17x_2 - 5x_3$$

$$6 - 12x_1 + 3x_2 = 42x_3 + 9 - 7x_1$$

在数学表示惯例上，所有包含变量（未知数）的量集中在方程的一边，并用方程中未知数的个数来称呼方程。上述的方程应该写为

$$5x = 4$$

$$2x_1 - 12x_2 = 8$$

$$-5x_1 + 3x_2 - 42x_3 = 3$$

并分别称为一元、二元和三元线性方程。一元、二元和n元线性方程的标准形式（standard form）分别是

$$ax = c$$

$$a_1x_1 + a_2x_2 = c$$

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = c$$

其中a为实数系数（coefficient），x为未知数（unknown）。

求解一个未知数的线性方程非常容易。如果有方程$ax=c$，则通过对方程两边除以a就可求得a: $x=c/a$（假定$a\neq 0$）。

具有两个未知数的线性方程却不太一样：一个解包含一对满足以下方程的数(x_1, x_2)

$$a_1x_1+a_2x_2=c$$

可以通过给x_1或x_2赋任意值（因而将方程的未知数减少为 1），然后像求解只含一个未知数的方程那样求解方程。例如，如果有方程

$$3x_1+2x_2=6$$

我们在方程中令$x_1=2$，则得到

$$\begin{aligned}3(2)+2x_2&=6\\6+2x_2&=6\\2x_2&=0\\x_2&=0\end{aligned}$$

因此，（2，0）就是一个解。然而，如果令$x_1=6$，则得到

$$\begin{aligned}3(6)+2x_2&=6\\18+2x_2&=6\\2x_2&=-12\\x_2&=-6\end{aligned}$$

这样就产生了另一个解$u=(6,-6)$。实际上，存在无数个解。在此，我们可以引入一些几何学上的直观解释：如果将变量x_1和x_2看成二维笛卡尔坐标系的x轴和y轴，则不同的各个解就可以表示平面上的一些点。在图 2.2 中，我们列举了一些解并画出了它们所表示的点。

含有两个未知数的线性方程的所有解的集合构成了一条直线，因此叫做“线性方程”。

x_1	x_2
−6	12
−4	9
−2	6
0	3
2	0
4	−3
6	−6

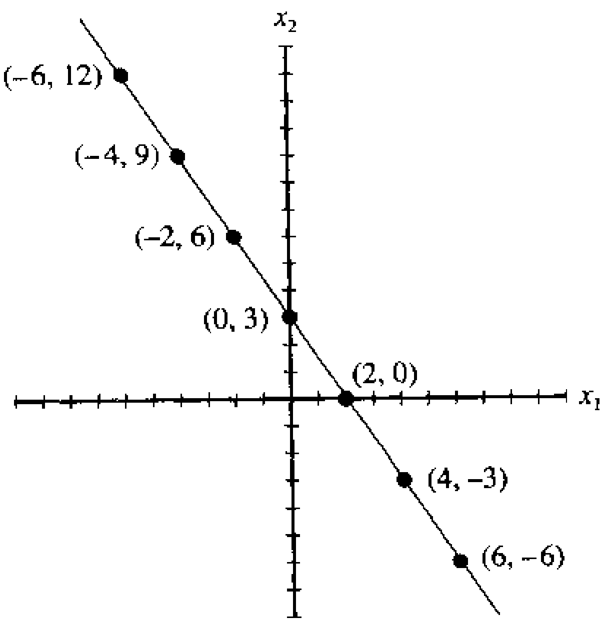


图 2.2　线性方程 $3x_1+2x_2=6$ 的解

2.4.2　两个未知数的线性系统

更加令人感兴趣也更有用的是线性系统（linear system），即由两个或多个线性方程所构成的线性方程组。我们将首先介绍由两个具有两个未知数的方程所构成的系统，其形式如下：

$$a_{1,1}x + a_{1,2}y = c_1$$

$$a_{2,1}x + a_{2,2}y = c_2$$

回想一下我们在前面讨论过的内容，具有两个未知数的方程的解可用于表示二维空间中的直线，因此，两个方程的线性系统就表示两条直线。即便不考虑下面的讨论，也可以简单地看出有三种情形需要考虑：

i．两条直线相交于一点。

ii．两条直线不相交——它们平行。

iii．两条直线重合。

回想一下具有两个未知数的单个线性方程的一个解表示一条直线上的一个点，第一种情形表明，存在一个点$u = (k_1, k_2)$，它是两个方程的解。在第二种情形中，不存在同时位于两条直线上的点，因此系统无解。在第三种情形中，存在无数的解，第一个方程所描述的线上的任何点同时也满足第二个方程（如图 2.3 所示）。当两个线性方程的系数成比例时将出现第二种和第三种情形：

$$\frac{a_{1,1}}{a_{2,1}} = \frac{a_{1,2}}{a_{2,2}}$$

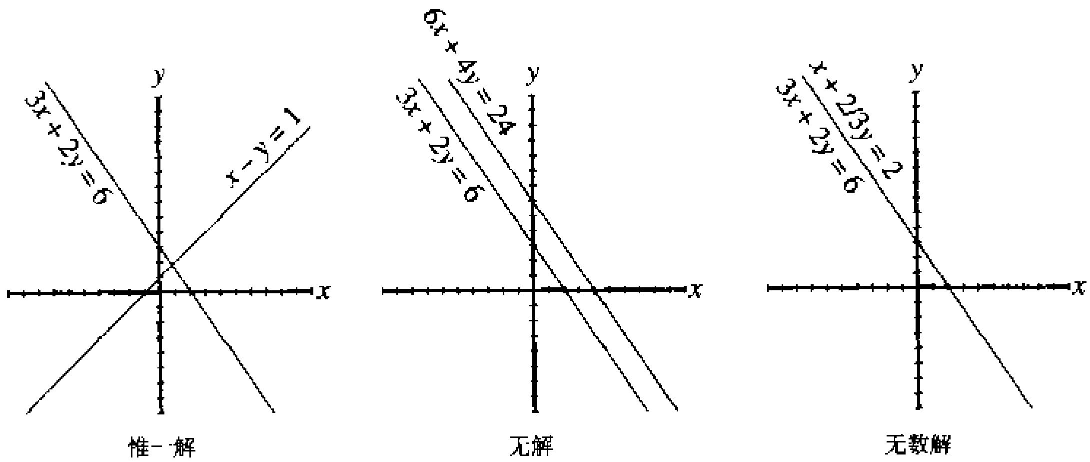


图 2.3　两个方程的线性系统的三组可能解

区分这两种情形的因素是常数项是否与系数成比例。如果系数和常数项都成比例，则系统具有无数个解（两条直线重合）：

$$\frac{a_{1,1}}{a_{2,1}} = \frac{a_{1,2}}{a_{2,2}} = \frac{c_1}{c_2}$$

如果系数成比例，而常数项不成比例，则系统无解（两条直线平行）：

$$\frac{a_{1,1}}{a_{2,1}} = \frac{a_{1,2}}{a_{2,2}} \neq \frac{c_1}{c_2}$$

如果只有一个解，则可用所谓的消元法（elimination）来求解：

第 1 步　用两个数分别乘以两个方程，使两个方程中其中一个未知数的系数互为相反数。

第 2 步　把得到的两个方程相加。这将消去一个未知数，得到一个只含一个未知数的方程。

第 3 步　求解该方程的一个未知数。

第 4 步　将求得的解代回原来的一个方程，得到一个新的只含一个未知数的方程。

第 5 步　求解另一个未知数。

【实例】给定

$$
\begin{aligned}
(1)\quad 3x + 2y &= 6 \\
(2)\quad x - y &= 1
\end{aligned}
$$

用 1 乘以（1），–3 乘以（2），并把它们相加：

$$
\begin{array}{rrcr}
1 \times (1): & 3x + 2y & = & 6 \\
-3 \times (2): & -3x + 3y & = & -3 \\
\hline
 & 5y & = & 3
\end{array}
$$

得到和：

于是很容易求得：$y = 3/5$。将它代回（1）可得

$$3x + 2\left(\frac{3}{5}\right) = 6$$

$$3x + \frac{6}{5} = 6$$

$$3x = \frac{24}{5}$$

$$x = \frac{8}{5}$$

因此，解为（8/5，3/5），与相交的点对应。■

2.4.3　一般线性系统

$m \times n$的线性方程系统的一般形式为

$$
\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= c_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= c_2 \\
&\vdots \\
a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= c_m
\end{aligned}
$$

$c_1 = c_2 = \cdots = c_m = 0$的系统称为齐次系统（homogeneous system）。线性系统常常写成矩阵形式：

$$\mathbf{AX} = \mathbf{C}$$

$$
\begin{bmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
 & & \vdots & \\
a_{m,1} & a_{m,2} & \cdots & a_{m,n}
\end{bmatrix}
\begin{bmatrix}
x_1 \\ x_2 \\ \vdots \\ x_n
\end{bmatrix}
=
\begin{bmatrix}
c_1 \\ c_2 \\ \vdots \\ c_m
\end{bmatrix}
$$

矩阵A叫做系数矩阵（coefficient matrix），而矩阵

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & c_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & c_2 \\ & & \vdots & & \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} & c_m \end{bmatrix}$$

称为增广矩阵（augmented matrix）。

求解一般线性系统的方法非常多，它们的通用性、复杂性、效率和稳定性各不相同。最常用的一种方法叫做高斯消元法（Gaussian elimination），其实就是上一节介绍的消元法的一般化扩展。高斯消元法算法的详细细节可在附录A的A.1节中找到。

2.4.4　减行、阶梯形和秩

我们再回过头来看看用消元法求解线性系统的例子，并将它表示为增广矩阵的形式：

$$\begin{bmatrix} 3 & 2 & 6 \\ 1 & -1 & 1 \end{bmatrix}$$

用–3 乘以第二行，这将产生一个等价方程对，其矩阵表示如下：

$$\begin{bmatrix} 3 & 2 & 6 \\ -3 & 3 & -3 \end{bmatrix}$$

下一步就是将两个方程相加在一起，并用得到的和代替其中的一个方程：

$$\begin{bmatrix} 3 & 2 & 6 \\ 0 & 1 & \frac{3}{5} \end{bmatrix}$$

然后走“捷径”，用$\frac{3}{5}$代回第一行并直接求解。

注意，矩阵的左下角元素为 0，这当然是由于我们为第二行所选择的乘数使得第一行与“缩放过”的第二行之和刚好能消去那个元素。

因此，应用这些运算，即用一个数乘以一行并用该行与另一行之和替代其中一行，显然不会影响系统的解。

可用于线性方程系统（或表示它们的矩阵）的另一种运算是换行，这也不会影响系统的解。显然，从数学的观点来看，方程的次序并不重要。

把这两种思想集成到一起，就从本质上描述了高斯消元法（参看A.1节）的一个基本思想：通过连续地消去所有行的第一个元素，最终将得到一个可通过回代来求解的系统。可是，这里讨论的重要方面是最终所得系统的形式（即回代前的形式）。原始的方程系统如下所示：

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + a_{1,3}x_3 + \cdots + a_{1,n}x_n &= c_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + a_{2,3}x_3 + \cdots + a_{2,n}x_n &= c_2 \\ &\vdots \\ a_{m,1}x_1 + a_{m,2}x_2 + a_{m,3}x_3 + \cdots + a_{m,n}x_n &= c_m \end{aligned}$$

最终得到的上三角形式如下：

$$
\begin{aligned}
b_{1,1}x_1 + b_{1,2}x_2 + b_{1,3}x_3 + \cdots + b_{1,n}x_n &= d_1 \\
b_{2,k_2}x_{k_2} + b_{2,k_3}x_3 + \cdots + b_{2,n}x_n &= d_2 \\
&\vdots \\
b_{r,k_r}x_{k_r} + \cdots + b_{r,n}x_n &= d_r
\end{aligned}
$$

注意，最后一个方程的下标不再与 m 有关，但是$r \leqslant m$，这是因为处理过程中将消去一些方程，处理过程中有时可能产生如下形式的方程：

$$0x_1 + 0x_2 + \cdots + 0x_n = c_i$$

如果$c_i = 0$，则方程可被完全消去而不会影响结果；如果$c_i \neq 0$，则方程是矛盾的（无解），从而可中止求解。结果是这些对行的连续运算将不断使系统变小。

关于数字 r 的若干其他的重要语句可表述如下：

- 如果$r = n$，则系统有惟一解。
- 如果$r < n$，则未知数的个数比方程多，这意味着系统有许多解。

一般地，我们把这些运算叫做基本行运算（elementary row operation）。基本行运算可应用于线性系统（以及其矩阵表示）而不会改变解集。这些运算可归纳如下：

- 交换两行。
- 用一个（非零）常数乘以一行。
- 用该行与另一行之和替代其中一行。

导致至少消去一个非零行元素的运算组合叫做减行（row reduction）。

由于运算应用于不同的方程，因此我们可以用对系统的矩阵表示的一系列变换来表示这些运算。作为该过程的最终结果（即一旦完成减行，矩阵不能进一步减行），方程系统和系统的矩阵表示都将变成阶梯形（echelon form）。这样的"完全减行"矩阵的方程个数 r 叫做矩阵的秩（rank）。因此，可以说，秩是矩阵的内在属性，只有在减行完成后才能显露出来。

基底（basis）、维数（dimension），以及线性无关（linear independence）等概念在如下方面与矩阵的秩相关：秩是矩阵的线性无关的行（或列）向量的个数，如果秩等于维数，则矩阵的行可作为矩阵所定义空间的基。

上述论断中，"秩等于阶梯形矩阵的线性无关行的数目"，来源于如下的事实：假设两个行向量不是线性无关的，即对于某些$a_1, a_2 \in \mathbb{R}$，有

$$a_1\vec{v}_i + a_2\vec{v}_j = 0$$

则将有一行可以进行合适的减行运算，而这意味着矩阵不是阶梯形的，与我们的假设矛盾。

2.5　方阵

在线性代数的一般应用领域中，方阵之所以显得特别重要，在很大程度上是由于它在表示、操作和求解线性系统方面所扮演的角色。在下一章中我们将会看到，这种重要性使得方阵在表示几何信息和进行几何变换方面也具有重要的地位。我们将从介绍几种特殊类型的方阵开始讨论。

2.5.1 对角矩阵

对角矩阵是指除对角线元素外的其他所有元素都为 0 的方阵：

$$\mathbf{M}=\begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix}$$

对角矩阵具有一些很有用的性质。

i．如果 **A** 和 **B** 是对角矩阵，则 $\mathbf{C}=\mathbf{AB}$ 也是对角矩阵。而且，计算 **C** 时，不需进行完整的矩阵计算，可以采用如下的高效方法：$c_{ii}=a_{ii}b_{ii}$，其他元素为 0。

ii．对角矩阵的乘法具有交换性：如果 **A** 和 **B** 是对角矩阵，则 $\mathbf{C}=\mathbf{AB}=\mathbf{BA}$ 。

iii．如果 **A** 是对角矩阵，**B** 是一般矩阵，并且 $\mathbf{C}=\mathbf{AB}$，则 **C** 的第 i 行等于 a_{ii} 乘以 **B** 的第 i 行；如果 $\mathbf{C}=\mathbf{AB}$，则 **C** 的第 i 列等于 a_{ii} 乘以 **B** 的第 i 列。

1．数量矩阵

数量矩阵是特殊类型的对角矩阵，其对角线上的元素都相同：

$$\mathbf{M}=\begin{bmatrix} \alpha & 0 & \cdots & 0 \\ 0 & \alpha & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \alpha \end{bmatrix}$$

2．单位矩阵

正如零矩阵是加法单位元，也存在一种类型的矩阵可以作为乘法单位元，通常简称为 **I**。因此，对任意矩阵 **M**，有 $\mathbf{IM}=\mathbf{MI}=\mathbf{M}$。注意，与零矩阵不同，单位矩阵不能是任意维的，它必须是方阵，因而有时记为 $\mathbf{I}_n$。对一个 $m\times n$ 矩阵 **M**，有 $\mathbf{I}_m\mathbf{M}=\mathbf{MI}_n=\mathbf{M}$。左上角至右下角对角线上的元素都为 1，其他元素都为 0 的矩阵就是单位矩阵。例如：

$$\mathbf{I}_2=\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \mathbf{I}_3=\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

一般地，单位矩阵的形式为：

$$\mathbf{I}_n=\begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

数量矩阵可看成数量与单位矩阵的乘积，即 $\alpha\mathbf{I}$。注意，与单位矩阵相乘等价于与数量 1 的数乘，并且与数量矩阵 $\alpha\mathbf{I}$ 相乘等价于与数量 α 的数乘。

2.5.2 三角形矩阵

两种特别重要的三角形矩阵分别称为上三角形矩阵和下三角形矩阵，它们的特点是，

除对角线之上或之下的元素外，其它元素都为 0：

$$\mathbf{M}=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{bmatrix}$$

$$\mathbf{M}=\begin{bmatrix} a_{1,1} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{bmatrix}$$

三角形矩阵也具有许多有用的性质：

i．如果 **A** 和 **B** 是下三角形矩阵，则 **C** = **AB** 也是下三角形矩阵。上三角形矩阵也具有类似性质。

ii．如果 **A** 和 **B** 是下三角形矩阵，则 **C** = **A** + **B** 也是下三角形矩阵。上三角形矩阵也具有类似性质。

iii．如果 **A** 是可逆的下三角形矩阵，则其逆矩阵 $\mathbf{A}^{-1}$ 也是下三角形矩阵。上三角形矩阵也具有类似性质（2.5.4 节介绍了有关逆矩阵的内容）。

正如在 2.4.4 和 A.1 节中所见到的，在表示和求解线性系统时，三角形矩阵特别重要。

2.5.3 行列式

方阵（对非方阵矩阵，没有行列式的概念）的行列式是一个实数，可以通过多种不同的方法来计算。对 2×2 和 3×3 的矩阵，行列式有许多直观的表示方法和用途（参见 3.3.2 和 4.4.4 节），但对于一般情形，不同的定义和计算方法就显得太随意。在二维空间中，一个矩阵 **M** 将一个顶点为 $\vec{0},\vec{\imath},\vec{\jmath},\vec{\imath}+\vec{\jmath}$ 的单位正方形映射成顶点为 $\vec{0},\vec{\imath}\mathbf{M},\vec{\jmath}\mathbf{M},(\vec{\imath}+\vec{\jmath})\mathbf{M}$ 的平行四边形。该平行四边形的面积为 $|\det(\mathbf{M})|$。如果 $\det(\mathbf{M})>0$，则 **M** 将保持正方形顶点的逆时针次序（即平行四边形的顶点次序也是逆时针方向的）；如果 $\det(\mathbf{M})<0$，则对应的平行四边形的顶点次序是顺时针方向的。在三维空间中，一个行列式不为零的矩阵 **M** 将单位立方体映射成体积为 $|\det(\mathbf{M})|$ 的平行六面体。顶点次序的保持方式与二维空间相似。

1．术语

我们首先介绍一些术语：给定一个矩阵 **M**，其行列式记为 $\det(\mathbf{M})$ 或 $|\mathbf{M}|$。“竖线”符号也经常用于矩阵标记。例如，如果有

$$\mathbf{M}=\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}$$

则可以将 $\det(\mathbf{M})$ 记为

$$|\mathbf{M}|=\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix}$$

2．计算 2×2 和 3×3 矩阵行列式的特殊方法

由于需要频繁使用，因此我们特地列出计算 2×2 和 3×3 矩阵行列式的公式：

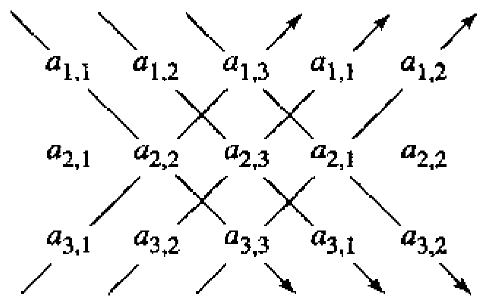
$$\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} = a_{1,1}a_{2,2} - a_{2,1}a_{1,2}$$

$$\begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} = \begin{matrix} a_{1,1}a_{2,2}a_{3,3} + a_{1,2}a_{2,3}a_{3,1} + \\ a_{1,3}a_{2,1}a_{3,2} - a_{3,1}a_{2,2}a_{1,3} - \\ a_{3,2}a_{2,3}a_{1,1} - a_{3,3}a_{2,1}a_{1,2} \end{matrix}$$

根据定义，1×1 矩阵的行列式公式为

$$|a_{1,1}| = a_{1,1}$$

事实上，经常需要用到这些公式，记住它们是非常有用的。记住 1×1 和 2×2 矩阵的行列式公式是很容易的，对 3×3 矩阵的行列式公式，有一个方便的诀窍，将矩阵写出来，并在矩阵的右边再写一遍矩阵的前两列；然后，把每一条对角线上的元素相乘在一起，并把所有从左上角到右下角对角线上的元素乘积相加，再减去所有从左下角到右上角对角线上的元素乘积：



注意，该方法也适用于 2×2 矩阵的行列式，但不适用于大于 3×3 矩阵的行列式。

3. 一般行列式的计算方法

计算 det(M) 的更通用方法叫做行列式的子式展开式（determinant expansion by minors）或拉普拉斯展开式（Laplacian expansion）。为了理解它，我们需要定义几个术语：子矩阵（submatrix）、子式（minor）和余子式（cofactor）。

子矩阵就是把一个矩阵的一行或多行和（或）一列或多列删除后所得的矩阵。例如，如果有矩阵

$$\mathbf{M} = \begin{bmatrix} 3 & 9 & 2 & 5 \\ 2 & 7 & 1 & 3 \\ 8 & 4 & 6 & 1 \\ 9 & 5 & 2 & 6 \end{bmatrix}$$

通过删除该矩阵 M 的第三列和第四行可得一个子矩阵：

$$\mathbf{M}'_{4,3} = \begin{bmatrix} 3 & 9 & 5 \\ 2 & 7 & 3 \\ 8 & 4 & 1 \end{bmatrix}$$

子式就是子矩阵的行列式。特别地，对于 M 的元素 $a_{i,j}$，其子式是通过删除 M 的第 i 行和第 j 列得到的矩阵 $\mathbf{M}'_{ij}$ 的行列式。

M 的元素 $a_{i,j}$ 的余子式 $c_{i,j}$ 就是该元素的子式或子式的相反数，可定义为

$$c_{i,j} = (-1)^{i+j}\left|\mathbf{M}'_{i,j}\right|$$

注意，这些余子式经常用于余子式矩阵，常记为 $\mathbf{C}$。下面用一个例子把它解释清楚。如果有一个 3×3 矩阵，可用如下基于余子式的方法来计算行列式：

$$\begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{vmatrix} = a_{1,1}\begin{vmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{vmatrix} - a_{1,2}\begin{vmatrix} a_{2,1} & a_{2,3} \\ a_{3,1} & a_{3,3} \end{vmatrix} + a_{1,3}\begin{vmatrix} a_{2,1} & a_{2,2} \\ a_{3,1} & a_{3,2} \end{vmatrix}$$

一般地，$n \times n$ 矩阵的行列式可“降阶”为 $(n-1)\times(n-1)$ 的行列式的倍数之和，可用同样的方法将降阶后的行列式再次降阶，直到得到单个的数为止（当然，在上述例子中，我们得到了 2×2 的子式，可用前面描述的直接方法来求解）。

4. 行列式的性质

像其他的矩阵运算一样，行列式也具有一些有趣并且有用的性质：

i．矩阵的行列式等于其转置矩阵的行列式：$|\mathbf{M}| = |\mathbf{M}^{\mathrm{T}}|$。

ii．两个矩阵之积的行列式等于两个矩阵的行列式之积：$|\mathbf{M}\mathbf{M}_1| = |\mathbf{M}|\,|\mathbf{M}_1|$。

iii．矩阵的逆矩阵的行列式等于矩阵行列式的倒数：$|\mathbf{M}^{-1}| = 1/|\mathbf{M}|$。

iv．单位矩阵的行列式等于 1：$|\mathbf{I}| = 1$。

v．数量与矩阵之积的行列式等于数量的矩阵大小次幂与矩阵的行列式之乘积，即：$|\alpha\mathbf{M}| = \alpha^n|\mathbf{M}|$。其中出现了 n，是因为 $\mathbf{M}$ 是 $n \times n$ 的矩阵。

vi．交换 $\mathbf{M}$ 的任意两行（或两列）将改变 $|\mathbf{M}|$ 的符号。

vii．如果用常数 α 乘以 $\mathbf{M}$ 的任意行（或列）的所有元素，则行列式将等于 $\alpha\,|\mathbf{M}|$。

viii．如果 $\mathbf{M}$ 的两行（或列）相同，则 $|\mathbf{M}| = 0$。

ix．三角形矩阵的行列式等于对角线上元素之积：

$$\begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ 0 & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{vmatrix} = \begin{vmatrix} a_{1,1} & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{vmatrix} = a_{1,1}a_{2,2}\cdots a_{n,n}$$

2.5.4 逆矩阵

我们已经看到数量乘法的许多性质都适用于矩阵。一个有用的性质是乘法逆元：对任意实数 $\alpha \neq 0$，存在一个数 β，使得 $\alpha\beta = 1$，当然 $\beta = 1/\alpha$。从上一节中我们看到，如果这一性质也适用于矩阵，那将非常有用。回忆一下，矩阵的单位元素是单位矩阵 $\mathbf{I}$，对于给定矩阵 $\mathbf{M}_1$，如果可能，我们希望找到矩阵 $\mathbf{M}_2$，使得 $\mathbf{M}_1\mathbf{M}_2 = \mathbf{I}$。如果存在这样的矩阵，则它就叫做 $\mathbf{M}_1$ 的逆矩阵，记为 $\mathbf{M}_2 = \mathbf{M}_1^{-1}$。

现在的问题是，如何计算 $\mathbf{M}_1^{-1}$？何时可能这样做？回想一下，矩阵 $\mathbf{M}_1$ 与 $\mathbf{M}_2$ 相乘是通过把 $\mathbf{M}_1$ 的第 i 行与 $\mathbf{M}_2$ 的第 j 列之积作为乘积矩阵的元素 i，j 而实现的。如果我们采用把 $\mathbf{M}_2$ 的每个 $n \times 1$ 的列记为 $\mathbf{v}_1, \mathbf{v}_2, \cdots, \mathbf{v}_n$ 的符号表示法，则乘积 $\mathbf{M}_1\mathbf{M}_2$ 可通过把 $\mathbf{M}_1$ 的每一行乘以 $\mathbf{M}_2$ 的列 $\mathbf{v}_i$，即 $\mathbf{M}_1\mathbf{v}_i$，来逐行计算。如果将单位矩阵 $\mathbf{I}$ 的每一列看成 $n \times 1$ 的向量 $\mathbf{e}_i$，除第 i 个元素为 1 外，它的其他元素都为 0，那么，我们可以将乘积 $\mathbf{M}_1\mathbf{M}_2 = \mathbf{I}$ 改写为：

$$\mathbf{M}_1[\mathbf{v}_1 \quad \mathbf{v}_2 \quad \cdots \quad \mathbf{v}_n] = [\mathbf{e}_1 \quad \mathbf{e}_2 \quad \cdots \quad \mathbf{e}_n]$$

可以将其理解为一系列的 n 阶线性系统：

$$\mathbf{M}_1\mathbf{v}_1 = \mathbf{e}_1$$

$$\mathbf{M}_1\mathbf{v}_2 = \mathbf{e}_2$$

$$\vdots$$

$$\mathbf{M}_1\mathbf{v}_n = \mathbf{e}_n$$

如果我们求解每一个 n 阶线性系统，将求得 $\mathbf{M}_2$ 的每一列，由于乘积为 $\mathbf{I}$，因此我们也就求得了 $\mathbf{M}_1^{-1}$。因为这些只是线性系统，所以我们可以采用任意的通用技术来求解，比如采用高斯消元法（参见 A.1 节）或 LU 分解法（Press 等，1988）。

另一种计算矩阵的逆矩阵的方法是通过逆矩阵的正式定义来求解：如果有方阵 $\mathbf{M}$，如果它存在逆矩阵 $\mathbf{M}^{-1}$，则 $\mathbf{M}^{-1}$ 的元素 $a_{i,j}^{-1}$ 定义为

$$a_{i,j}^{-1} = \frac{(-1)^{i+j}\left|\mathbf{M}'_{j,i}\right|}{|\mathbf{M}|}$$

回想一下，该式的分子就是一个余子式，因此我们可以将它写成

$$\mathbf{M}^{-1} = \frac{\mathbf{C}^{\mathrm{T}}}{|\mathbf{M}|}$$

注意，式中使用转置是因为前式中的下标次序是 j，i。$\mathbf{C}^{\mathrm{T}}$ 也叫做 $\mathbf{M}$ 的伴随矩阵（adjoint）。

为了说明上式是如何求得的，我们举一个 2×2 矩阵的简单例子。设

$$\mathbf{M} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

其行列式 $|\mathbf{M}| = 1 \times 4 - 3 \times 2 = -2$。再计算余子式

$$c_{11} = (-1)^{1+1}|\mathbf{M}'_{11}| \;\; = |4| = 4$$

$$c_{12} = (-1)^{1+2}|\mathbf{M}'_{12}| = -|3| = -3$$

$$c_{21} = (-1)^{2+1}|\mathbf{M}'_{21}| = -|2| = -2$$

$$c_{22} = (-1)^{2+2}|\mathbf{M}'_{22}| \;\; = |1| = 1$$

可得

$$\mathbf{C} = \begin{bmatrix} 4 & -3 \\ -2 & 1 \end{bmatrix}$$

于是逆矩阵为

$$\mathbf{M}^{-1} = \frac{\mathbf{C}^{\mathrm{T}}}{|\mathbf{M}|}$$

$$= \frac{\begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix}}{-2}$$

$$=\begin{bmatrix}-2 & 1\\ 3/2 & -1/2\end{bmatrix}$$

校验一下，有

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}\begin{bmatrix}-2 & 1\\ 3/2 & -1/2\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=\mathbf{I}$$

1. 逆矩阵的性质

矩阵的逆矩阵（假定逆矩阵存在）具有许多有用的性质：

i. 如果 $\mathbf{M}\mathbf{M}^{-1}=\mathbf{I}$，则 $\mathbf{M}^{-1}\mathbf{M}=\mathbf{I}$

ii. $(\mathbf{M}_1\mathbf{M}_2)^{-1}=\mathbf{M}_2^{-1}\mathbf{M}_1^{-1}$

iii. $(\mathbf{M}^{-1})^{-1}=\mathbf{M}$

iv. $(\alpha\mathbf{M})^{-1}=(1/\alpha)\mathbf{M}^{-1}\ (\alpha\neq 0)$

2. 逆矩阵何时存在？

前面几节已经暗示方阵的逆矩阵并不总是存在的，确实如此。那么，问题是，如何确定给定的 $n\times n$ 矩阵 $\mathbf{M}$ 的逆矩阵是否存在呢？有几种方法（它们都是等价的）可以做到这一点：

i. 其秩为 n。

ii. 它是非奇异的。

iii. $|\mathbf{M}|\neq 0$。

iv. 它的行（列）是线性无关的。

v. 把它作为一个变换，并不会减少维数。

3. 奇异矩阵

如果矩阵 $\mathbf{M}$ 是一个方阵（$n\times n$）并且其行列式的值为非零（$|\mathbf{M}|\neq 0$），则称矩阵 $\mathbf{M}$ 为非奇异（nonsingular）的。任何不同时满足上述条件的矩阵都称为奇异的（singular）。非奇异性是矩阵的重要性质，如下的判断矩阵 $\mathbf{M}$ 的非奇异性的充要条件的等价列表说明了这一点：

i. $|\mathbf{M}|\neq 0$。

ii. 其秩为 n。

iii. 矩阵 $\mathbf{M}^{-1}$ 存在。

iv. 齐次系统 $\mathbf{MX}=\mathbf{0}$ 仅有一个简单解 $\mathbf{X}=\mathbf{0}$。

最后一项特别重要，因为该性质和第一个充要条件说明，当且仅当 $|\mathbf{M}|=0$ 时，齐次系统 $\mathbf{MX}=\mathbf{0}$ 有一个非简单解（$\mathbf{X}\neq\mathbf{0}$）。

2.6 线性空间

在本节中，我们将介绍线性（或向量）空间的概念，并基于矩阵和矩阵运算来讨论向量空间的表示和线性空间中的运算。与一些介绍该主题的方法不同，我们将暂不提及线性空间的任何类型的几何解释；在随后的一章中，我们先用抽象的方式完整地介绍几何向量，

然后再明确地探讨这些问题。

2.6.1 数域

在正式定义线性空间之前，我们需要先定义术语数域（field）。数域是“一个代数系统，对其中的元素执行加、减、乘和除（被零除除外）运算所得结果依然属于该系统，并且具有结合律、交换律和分配律”（*www.wikipedia.com/wiki/Field*）。形式上，数域 F 包含一个集合和两个具备如下性质的二元运算符“+”和“*”（加和乘）。

i．F 具有加法和乘法运算的封闭性：$\forall a,b \in F$，则有 $(a+b) \in F$ 和 $(a*b) \in F$。

ii．加法和乘法运算的结合性：$\forall a,b,c \in F$，则有 $a+(b+c)=(a+b)+c$ 和 $a*(b*c)=(a*b)*c$。

iii．加法和乘法运算的交换性：$\forall a,b \in F$，则有 $a+b=b+a$ 和 $a*b=b*a$。

iv．乘法对加法的分配性：$\forall a,b,c \in F$，则有 $a*(b+c)=(a*b)+(a*c)$ 和 $(b+c)*a=(b*a)+(c*a)$。

v．存在加法单位元：$\exists 0 \in F$，使得 $\forall a \in F, a+0=a$ 和 $0+a=a$。

vi．存在乘法单位元：$\exists 1 \in F$，使得 $\forall a \in F, a*1=a$ 和 $1*a=1$。

vii．加法逆元：$\forall a \in F, \exists -a \in F$，使得 $a+(-a)=0$ 和 $(-a)+a=0$。

viii．乘法逆元：$\forall a \neq 0 \in F, \exists a^{-1} \in F$，使得 $a*a^{-1}=1$ 和 $a^{-1}*a=1$。

数域也叫做交换环（commutative ring）或交换除法代数（commutative division algebra）。数域的例子有：

- 有理数 $\mathbb{Q}=\{\frac{a}{b} | a,b \in \mathbb{Z}, b \neq 0\}$，其中 $\mathbb{Z}$ 表示整数
- 实数 $\mathbb{R}$
- 复数 $\mathbb{C}$

注意，整数不形成一个数域，只是一个环（整数不存在乘法逆元）。

2.6.2 定义和性质

非正式地说，线性空间包含一组对象（叫做向量）、实数（数量）和两种运算（向量相加和向量与数相乘），这两种运算需要具有一些特定的性质。不再局限于采用黑体小写字母这一基本的多元组表示法，我们使用一种表示法，它可明确地表明我们在处理向量这一事实，即将向量表示为带区别箭头的斜体小写字母。通常，我们使用 $\vec{u}, \vec{v}$ 和 $\vec{w}$，或 $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ 来表示向量。在形式上，假设有：

- 一个数域 K（对我们来说，就是 $\mathbb{R}$）。
- 一个（非空）向量集 $\mathcal{V}$。
- 对元素 $\vec{u}, \vec{v} \in \mathcal{V}$ 定义加法运算符“+”。
- 对数量 $k \in K$ 和元素 $\vec{v} \in \mathcal{V}$ 定义乘法运算符“*”（通常省略“*”而使用连接式 $\vec{v}=k\vec{u}$）。
- 加法和乘法具有如下所列的性质。

i．乘法具有封闭性：$\forall k \in K$ 和 $\forall \vec{v} \in \mathcal{V}$，则 $k\vec{v} \in \mathcal{V}$。

ii．加法具有封闭性：$\forall \vec{u}, \vec{v} \in \mathcal{V}$，则 $\vec{u}+\vec{v} \in \mathcal{V}$。

iii. 加法结合性：$\forall \vec{u}, \vec{v}, \vec{w} \in \mathcal{V}$，则 $\vec{u} + (\vec{v} + \vec{w}) = (\vec{u} + \vec{v}) + \vec{w}$ 。

iv. 存在加法单位元素：$\forall \vec{v} \in \mathcal{V}$，$\exists$ 向量 $\vec{0} \in \mathcal{V}$，叫做零向量，使得 $\vec{v} + \vec{0} = \vec{v}$。

v. 存在加法逆元：$\forall \vec{v} \in \mathcal{V}$，$\exists$ 向量 $-\vec{v}$，使得 $\vec{v} + (-\vec{v}) = \vec{0}$ 。

vi. 加法结合性：$\forall \vec{u}, \vec{v} \in \mathcal{V}$，则 $\vec{u} + \vec{v} = \vec{v} + \vec{u}$ 。

vii. 乘法对加法的分配性：$\forall k \in K$ 和 $\forall \vec{u}, \vec{v} \in \mathcal{V}$，则 $k(\vec{u} + \vec{v}) = k\vec{u} + k\vec{v}$ 。

viii. 加法对乘法的分配性：$\forall k_1, k_2 \in K$ 和 $\forall \vec{v} \in \mathcal{V}$，则 $(k_1 + k_2)\vec{v} = k_1\vec{v} + k_2\vec{v}$ 。

ix. 乘法的结合性：$\forall k_1, k_2 \in K$ 和 $\forall \vec{v} \in \mathcal{V}$，则 $(k_1k_2)\vec{v} = k_1(k_2\vec{v})$。

x. 存在乘法单位元：$\forall \vec{v} \in \mathcal{V}$，则 $1 * \vec{v} = \vec{v}$ 。

如前面所述，我们这里关心的是计算机图形学，因此，数域 K 就是实数域 $\mathbb{R}$，$\mathcal{V}$ 中的向量就是实数多元组：$\mathbf{a} = (a_1, a_2, \ldots, a_n)$。在后面的章节中，一旦我们建立起几何向量、向量空间和矩阵之间的关系，我们将不再使用这种相对抽象的面向多元组的向量表示法，转而使用一种能反映 $\mathbb{R}^n$（所有 n 维多元组的集合）中多元组的几何意义的表示方法。

2.6.3 子空间

给定 $\mathbb{R}$ 上的线性空间 $\mathcal{V}$，设 S 为 $\mathcal{V}$ 的子集，并设 S 和 $\mathcal{V}$ 的运算相同。如果 S 也是 $\mathbb{R}$ 上的线性空间，则 S 是 $\mathcal{V}$ 的子空间（subspace）。

上述定义非常明确，其精妙之处在于，它指出了线性空间的子集可能是也可能不是线性空间。Agnew 和 Knapp 提供的例子非常恰当地说明了这点：考虑 $\mathbb{R}^3$ 的子集 S_1，它由形式为 $(a_1, a_2, 0)$ 的所有三维多元组组成。快速地用定义线性空间的规则比照该子集，可以发现它确实是一个线性空间。然而，假设有一个由形式为 $(a_1, a_2, 1)$ 的所有三维多元组组成的子集 S_2，并检查是否线性空间的所有规则都适合它，那么，我们将发现，如下几条规则并不适合该子集。

i. 加法具有封闭性：$(a_1, a_2, 1) + (b_1, b_2, 1) = (a_1 + b_1, a_2 + b_2, 2)$，结果不属于 S_2 。

ii. 乘法具有封闭性：$(a_1, a_2, 1) \in S_2$，但是对所有 $k \neq 1$，有 $k(a_1, a_2, 1) = (ka_1, ka_2, k) \notin S_2$ 。

iii. 存在加法单位元素：$(0, 0, 0) \notin S_2$

iv. 存在加法逆元：$(a_1, a_2, 1) \in S_2$，但是 $(-a_1, -a_2, -1) \notin S_2$。

注意该例并不具备任意性，这很有意思，其重要性将在后面凸显出来。

2.6.4 线性组合和生成空间

项目集合的线性组合（linear combination）由项目的数乘之和构成。假设有一个向量集 $\mathcal{A}$，其元素之和是 $\mathbb{R}^n$ 中的一个向量（多元组）集合：$\{\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n\}$。可以生成向量 $\vec{u} = k_1\vec{a}_1 + k_2\vec{a}_2 + \cdots + k_n\vec{a}_n$ 。向量 $\vec{u}$ 本身是一个向量，因为每一个 $k_i\vec{a}_i$ 都是向量，它们之和当然也是一个向量。

给定一个向量（多元组）集合 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$，它定义了一个线性空间 $\mathcal{V}$，向量的所有线性组合的集合 S 本身就是一个线性空间，并且该空间是由 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 生成（spanned）的空间。集合 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 叫做 S 的生成集合（spanning set）。生成集合这一概念的重要性体现在：任何向量 $\vec{w} \in S$ 都可通过 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 来表示，只需找到数量

$k_1, k_2, \ldots, k_n$，使得 $\vec{w} = k_1\vec{v}_1 + k_2\vec{v}_2 + \cdots + k_n\vec{v}_n$。

2.6.5 线性无关、维数和基底

线性组合这一概念在线性代数中经常使用，对于理解线性无关和基底（它们是直观地理解线性代数的两个要点），它显得尤为重要。

1．线性无关

假设有一个向量空间 $\mathcal{V}$，对于任何向量集合 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$，它们不是线性无关的就是线性相关的。如果存在不全为 0 的常数 $c_1, c_2, \ldots, c_n$，使得

$$c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_n\vec{v}_n = \vec{0}$$

则称该向量集合定义为线性相关的。而如果

$$c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_n\vec{v}_n = \vec{0}$$

仅当所有常数都为 0 时才成立，则称该向量集合为线性无关的。

下面是一个线性相关的向量集合的例子：$\vec{v}_1 = (2, 5, 3), \vec{v}_2 = (1, 4, 0), \vec{v}_3 = (7, 22, 6)$，这时由于常数集合 $\{2, 3, -1\}$ 导致如下的结果：

$$\begin{aligned} 2\vec{v}_1 + 3\vec{v}_3 + -1\vec{v}_3 &= 2(2, 5, 3) + 3(1, 4, 0) + -1(7, 22, 6) \\ &= (4, 10, 6) + (3, 12, 0) + (-7, -22, -6) \\ &= \vec{0} \end{aligned}$$

上述线性相关的定义是相当标准的，但可能不是最直观的，特别是对于我们的目标（在下一章中该目标将很清楚）而言。当且仅当任何向量都不是其他向量的线性组合时，向量集合 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 中的非零向量才是线性无关的。在上面的例子中，$\vec{v}_3$ 是 $\vec{v}_1$ 和 $\vec{v}_2$ 的线性组合，其中系数分别为 2 和 3。实际上，可以下一个更严格的定义：当且仅当向量集合 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 中的任何一个向量 $\vec{v}_i$ 都是排在它前面的向量的一个线性组合时，称该向量集合中的非零向量是线性相关的：

$$c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_{i-1}\vec{v}_{i-1} = \vec{v}_i$$

2．基底和维数

在定义空间的维数时，线性无关的概念至关重要。如果有向量集合 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$，当且仅当它们是线性无关的并生成该空间时，它们将构成线性空间 $\mathcal{V}$ 的一个基底。$\mathcal{V}$ 的维数（dimension）是 n，即线性无关的向量的个数。

从定义中可以得出如下的结论：

- $\mathcal{V}$ 中向量个数小于 n 的任意线性无关向量的集合都不能生成 $\mathcal{V}$。
- $\mathcal{V}$ 中向量个数大于 n 的任意向量的集合都是线性相关的。
- 维数为 n 的空间 $\mathcal{V}$ 不存在惟一的基底；存在无数的元素个数为 n 的基底向量集合。

子空间、生成空间、线性组合和维数以如下方式联系在一起：假定 $\mathcal{V}$ 是维数为 n 的向量空间，由基底向量 $\mathrm{V} = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 所生成，且定义为该基底向量的所有线性组合，那么，如果选择线性无关向量集合 $\mathrm{W} = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_m\} \in \mathcal{V}$，其中 $m < n$，那么，W 的所有线性组合向量的集合构成 $\mathcal{V}$ 的维数为 n 的子空间 $\mathcal{W}$。

2.7 线性映射

在本节中，我们将首先复习一下映射的基础概念，并在此基础上进一步介绍线性映射，即从一个线性空间到另一个线性空间的函数。然后我们将说明矩阵是如何用于表示线性映射的。

2.7.1 映射基础

函数的基本概念是联系一个集合的元素与另一个集合的元素的规则。术语映射（mapping）、函数（function）和变换（transformation）都是用来说明这样的元素对之间的特殊关系的同义词。

【定义】假定 $\mathcal{A}$ 和 $\mathcal{B}$ 是元素分别为 $\{a_1, a_2, \ldots, a_m\}$ 和 $\{b_1, b_2, \ldots, b_n\}$ 的两个集合，则从 $\mathcal{A}$ 到 $\mathcal{B}$ 的函数，记为：

$$T : \mathcal{A} \longrightarrow \mathcal{B}$$

它是元素对 (a, b) 的集合，其中 $a \in \mathcal{A}$，$b \in \mathcal{B}$。集合中每一个元素对都是惟一的，并且每一个元素 $a \in \mathcal{A}$ 都必定出现在集合的一个元素对中。集合 $\mathcal{A}$ 叫做函数的定义域（domain），集合 $\mathcal{B}$ 叫做值域（range）或上域（co-domain）。函数的示意图如图 2.4 所示。■

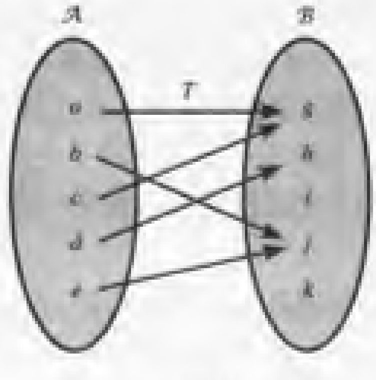


图 2.4　函数的示意图

对每一个元素 $a \in \mathcal{A}$，$\mathcal{B}$ 中通过函数与 a 相关联的值叫做 a 的映像（image），记为 $T(a)$ 或 aT。如果一个元素 $b \in \mathcal{B}$ 是一些元素 $a \in \mathcal{A}$ 的映像，则 a 叫做 b 的原像（preimage）。理解每一个元素 $a \in \mathcal{A}$ 都会出现在集合的一个元素对中是很重要的，但并不需满足每一个元素 $b \in \mathcal{B}$ 都会出现。实际上，有的函数可能将 $\mathcal{A}$ 中的每一个元素全都映射到 $\mathcal{B}$ 中的同一个元素上。

函数的定义域和值域可以是任何对象的集合，其中的元素可以是相同类型的，也可以是不同类型的。我们可以将实数映射到实数，例如，T 可以被定义为实数到其平方根的映射：$x \mapsto \sqrt{x}$。

1. 映射组合

假定有两个函数：$T:\mathcal{A}\longrightarrow\mathcal{B}$ 和 $U:\mathcal{B}\longrightarrow\mathcal{C}$。根据定义，对每一个 $a\in\mathcal{A}$，都存在一些 $b\in\mathcal{B}$，使得 $T(a)=b$。同样，函数 U 将元素 b（a 的映像）映射到一些元素 $c\in\mathcal{C}$。两个函数 T 和 U 对 a 的映射叫做 T 和 U 的组合（composition），记为

$$(U\circ T)(a)=U(T(a))$$

（或采用另一种记法：$a(T\circ U)=aTU$），并且认为

$$a\mapsto U(T(a))$$

映射组合具有结合性。假定有三个函数：$T:\mathcal{A}\longrightarrow\mathcal{B}$，$U:\mathcal{B}\longrightarrow\mathcal{C}$ 和 $V:\mathcal{C}\longrightarrow\mathcal{D}$，则 $(V\circ U)\circ T=V\circ(U\circ T)$。两个函数的组合示意如图 2.5 所示。

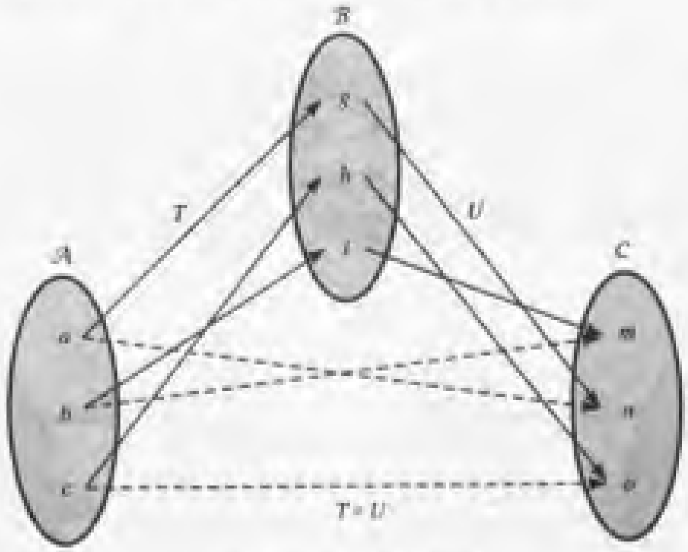


图 2.5　两个函数的组合

2. 特殊类型的映射

一对一映射（one-to-one）、映成（onto）和同构映射（isomorphic，既是一对一映射，又是映成）是三种重要的映射类型。它们的示意显示如图 2.6 所示。一对一映射 $T:\mathcal{A}\longrightarrow\mathcal{B}$ 是指每一个 $a\in\mathcal{A}$ 都与惟一的一个 $b\in\mathcal{B}$ 相关联。映成 $T:\mathcal{A}\longrightarrow\mathcal{B}$ 是指每一个 $b\in\mathcal{B}$ 都是某些 $a\in\mathcal{A}$ 的映像。例如，这类函数有 $T(x)=2^x$ 和 $T(x)=x^2$，$\mathcal{A}$ 为实数集合，对前者，$\mathcal{B}$ 是正实数的集合，对后者，$\mathcal{B}$ 是非负实数的集合（如图 2.7 所示）。

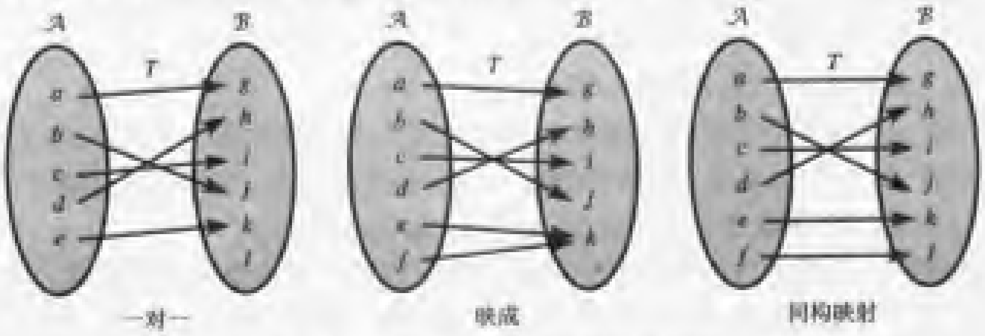
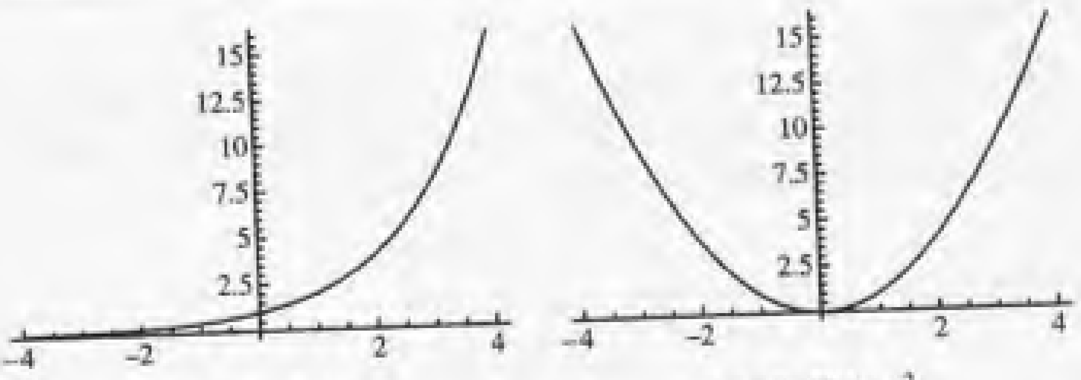


图 2.6　一对一映射、映成和同构映射



一对一函数：$T(x) = 2^x$　　　　映成函数：$T(x) = x^2$

图 2.7　一对一函数和映成函数

3．逆映射

给定一个映射 $T: \mathcal{A} \longrightarrow \mathcal{B}$，我们会很自然地想到与 T 相逆的映射。一般地，如果存在一个映射 $T^{-1}: \mathcal{B} \longrightarrow \mathcal{A}$，使得 $TT^{-1} = I$，则线性映射 T 是可逆的，其中 I 是单位映射。由于 T^{-1} 是一个映射，因此，根据定义，它必定具有 $\mathcal{B}$ 的全集作为其定义域：每一个元素 $a \in \mathcal{A}$ 必定在 T^{-1} 的值域中。这两个事实说明，只有同时是一对一映射和映成的映射才是可逆的。如图 2.8 所示。

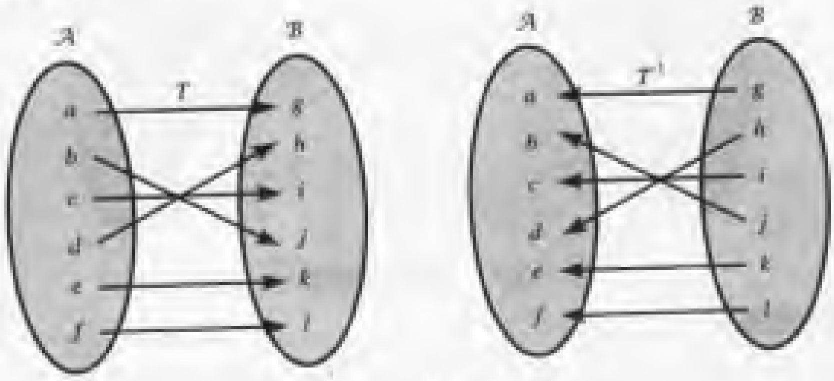


图 2.8　一个可逆映射

2.7.2　线性映射

当然，我们真正感兴趣的是线性映射，即与线性空间相关的映射。给定两个线性空间 $\mathcal{A}$ 和 $\mathcal{B}$，线性映射 $T: \mathcal{A} \longrightarrow \mathcal{B}$ 是一个保留向量加法和数量乘法的函数：

i．$\forall \vec{u}, \vec{v} \in \mathcal{A}, T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v})$

ii．$\forall \alpha \in \mathbb{R}$ and $\forall \vec{v} \in \mathcal{A}, T(\alpha\vec{v}) = \alpha T(\vec{v})$

该定义的一个重要含义是线性映射保留了线性组合：即 $T(\alpha\vec{u} + \beta\vec{v}) = \alpha T(\vec{u}) + \beta T(\vec{v})$。

在前一节中，我们已提到，映射可能是一对一映射或映成。如果线性函数 $T: \mathcal{A} \longrightarrow \mathcal{B}$ 将 $\mathcal{A}$ 一对一映射且映成到 $\mathcal{B}$，则它是一个同构（isomorphism）。

线性映射的一个重要方面是，它们是完全由它们进行变换基底向量的方式来确定的，通过复习如下的内容，就能理解这一点：任何向量 $\vec{v} \in \mathcal{V}$ 都可表示为基底向量的线性组合，而线性映射保留了线性组合。

2.7.3 线性映射的矩阵表示

可用矩阵来表示从 $\mathbb{R}^m$ 到 $\mathbb{R}^n$ 的线性映射，即我们可用矩阵来说明 $\mathcal{A}$ 中的向量是怎样映射到 $\mathcal{B}$ 中的向量的。我们已经知道，线性映射是完全由它对基底向量的变换效果所确定的，这一事实也将说明矩阵是如何用于定义（或实现）线性变换的。

假定有线性空间 $\mathcal{A}$ 和 $\mathcal{B}$，它们的基底向量分别为 $\vec{u}_1,\vec{u}_2,\ldots,\vec{u}_m$ 和 $\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n$，并有一个线性映射 $T:\mathcal{A}\longrightarrow\mathcal{B}$。被变换的基底 $T(\vec{u}_1),T(\vec{u}_2),\ldots,T(\vec{u}_m)$ 是 $\mathcal{B}$ 的元素，因而可用 $\mathcal{B}$ 的基底向量 $\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n$ 的一些线性组合来表示：

$$T(\vec{u}_1)=a_{1,1}\vec{v}_1+a_{1,2}\vec{v}_2+\cdots+a_{1,n}\vec{v}_n$$

$$T(\vec{u}_2)=a_{2,1}\vec{v}_1+a_{2,2}\vec{v}_2+\cdots+a_{2,n}\vec{v}_n$$

$$\vdots$$

$$T(\vec{u}_m)=a_{m,1}\vec{v}_1+a_{m,2}\vec{v}_2+\cdots+a_{m,n}\vec{v}_n$$

我们可以得到上式的系数矩阵 **T**，它就是与 $\mathcal{A}$ 和 $\mathcal{B}$ 的基底相关的线性映射 T 的矩阵表示：

$$\mathbf{T}=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

由此可得出两个重要的事实：

1. $\mathcal{A}$ 中任何向量的（行）矩阵表示与 **T** 相乘，可将它变换到空间 $\mathcal{B}$ 中：

$$T(\vec{x})=[\,x_1 \quad x_2 \quad \cdots \quad x_m\,]\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}$$

2. 两个线性映射的组合的矩阵表示就是每一个映射的矩阵表示的连接。设 $a_1,a_2,\ldots,a_m$，$b_1,b_2,\ldots,b_n$ 和 $c_1,c_2,\ldots,c_l$ 分别是线性空间 $\mathcal{A}$，$\mathcal{B}$ 和 C 的基底，设 $T:\mathcal{A}\longrightarrow\mathcal{B}$ 和 $S:\mathcal{B}\longrightarrow C$ 是矩阵表示分别为 **T** 和 **S** 的线性映射。则 T 和 S 的组合 $R:\mathcal{A}\longrightarrow C$ 可表示如下：

$$S(T(\vec{v}))=\vec{v}\,\mathbf{TS}$$

2.7.4 克莱姆定理

如果线性方程组系统有一个解，则克莱姆定理是进行直接求解的一种方法。为了说明该定理，假定有如下的具有两个变量的线性系统：

$$a_{1,1}x_1+a_{1,2}x_2=c_1$$

$$a_{2,1}x_1+a_{2,2}x_2=c_2$$

如果使用消元法，将第一个方程乘以 $a_{2,1}$，将第二个方程乘以 $a_{1,1}$，然后相减，可得：

$$\begin{array}{rcl} a_{2,1}a_{1,1}x_1 + a_{2,1}a_{1,2}x_2 & = & a_{2,1}c_1 \\ a_{1,1}a_{2,1}x_1 + a_{1,1}a_{2,2}x_2 & = & a_{1,1}c_2 \\ \hline a_{2,1}a_{1,2}x_2 - a_{1,1}a_{2,2}x_2 & = & a_{2,1}c_1 - a_{1,1}c_2, \end{array}$$

即

$$x_2 = \frac{a_{1,1}c_2 - a_{2,1}c_1}{a_{1,1}a_{2,2} - a_{2,1}a_{1,2}}$$

倘若$a_{1,1}a_{2,2} - a_{2,1}a_{1,2} \neq 0$，将该值在第一个方程中回代$x_2$，得

$$a_{1,1}x_1 + a_{1,2}x_2 = c_1$$

$$a_{1,1}x_1 + x_2\left(\frac{a_{1,1}c_2 - a_{2,1}c_1}{a_{1,1}a_{2,2} - a_{2,1}a_{1,2}}\right) = c_1$$

$$a_{1,1}x_1 = c_1 - \frac{a_{1,2}a_{1,1}c_2 - a_{1,2}a_{2,1}c_1}{a_{1,1}a_{2,2} - a_{1,2}a_{2,1}}$$

$$= \frac{a_{1,1}a_{2,2}c_1 - a_{1,2}a_{1,1}c_2}{a_{1,1}a_{2,2} - a_{1,2}a_{2,1}}$$

$$x_1 = \frac{c_1a_{2,2} - c_2a_{1,2}}{a_{1,1}a_{2,2} - a_{1,2}a_{2,1}}$$

倘若$a_{1,1}a_{2,2} - a_{2,1}a_{1,2} \neq 0$，$x_1$和$x_2$的分子和分母都可以表示成行列式：

$$x_1 = \frac{\begin{vmatrix} c_1 & a_{1,2} \\ c_2 & a_{2,2} \end{vmatrix}}{\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix}}, \quad x_2 = \frac{\begin{vmatrix} a_{1,1} & c_1 \\ a_{2,1} & c_2 \end{vmatrix}}{\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix}}$$

我们可以用克莱姆定理来求解 2.4.2 节的例子。方程为

$$3x + 2y = 6$$

$$x - y = 1$$

应用克莱姆定理，可得

$$x_1 = \frac{\begin{vmatrix} 6 & 2 \\ 1 & -1 \end{vmatrix}}{\begin{vmatrix} 3 & 2 \\ 1 & -1 \end{vmatrix}} = \frac{8}{5}$$

$$x_2 = \frac{\begin{vmatrix} 3 & 6 \\ 1 & 1 \end{vmatrix}}{\begin{vmatrix} 3 & 2 \\ 1 & -1 \end{vmatrix}} = \frac{3}{5}$$

克莱姆定理的一般形式如下：设A表示系数矩阵

$$\mathbf{A} = [a_{i,j}]$$

并设$\mathbf{B}_i$为用常数$c_1, c_2, \cdots, c_n$替换$\mathbf{A}$的第i列所得的矩阵。那么，如果$|\mathbf{A}| \neq 0$，则存在惟

一解：

$$\vec{u} = \left(\frac{|\mathbf{B}_1|}{|\mathbf{A}|}, \frac{|\mathbf{B}_2|}{|\mathbf{A}|}, \cdots, \frac{|\mathbf{B}_n|}{|\mathbf{A}|}\right)$$

应该注意，与其他方法一样，当矩阵的行列式接近零时，克莱姆定理可能变得不稳定。将克莱姆定理应用于大型系统时存在两个问题：第一，效率，对于$n \times n$的系统，克莱姆定理的计算量是$O(n!)$，而高斯消元法是$O(n^3)$；第二，减法误差，这是高斯消元法所重点处理的问题。

2.8 特征值和特征向量

回忆一下，我们把$n \times 1$和$1 \times n$的矩阵叫做向量，无需对此着墨太多（在下一章中我们将讨论线性代数和向量的几何意义），可以仅仅将向量想像为在一些“空间”（比如笛卡尔三维空间）中指定的方向和距离。因此，用矩阵乘以一个向量就可以被认为是变换向量的方向和/或长度（注意，我们现在又将讨论局限于方阵）。

我们可用常见的形式来表示向量 $\vec{v}$ 和矩阵 **M** 的这种乘法：

$$\vec{v}' = \vec{v}\mathbf{M}$$

对某些特殊的向量 $\vec{v}$，可以找到一个常数λ，使得

$$\vec{v}' = \lambda\vec{v}$$

因而有

$$\vec{v}\mathbf{M} = \lambda\vec{v}$$

可以找到这样的值λ的向量 $\vec{v}$ 叫做 **M** 的特征向量（eigenvector），值λ叫做特征值（eigenvalue）。注意，由于λ是一个数量，值 $\lambda\vec{v}$ 就是 $\vec{v}$ 的一个缩放版本，因此，无论 **M** 怎样与任意向量相乘，结果都只不过是缩放其特征向量而已。

现在产生的疑问是：如何寻找特定矩阵 **M** 的特征值？我们对定义稍做推算：

$$\vec{v}\mathbf{M} = \lambda\vec{v} \tag{2.2}$$

$$\vec{v}(\lambda\mathbf{I} - \mathbf{M}) = \vec{0} \tag{2.3}$$

只要 $\vec{v}$ 不是 $\vec{0}$ 向量，则必定有

$$|\lambda\mathbf{I} - \mathbf{M}| = 0$$

该式叫做特征多项式（characteristic polynomial）。

下面是一个例子：设

$$\mathbf{M} = \begin{bmatrix} 6 & 4 \\ 2 & 4 \end{bmatrix}$$

我们要寻找数量λ，使得 $\vec{v}\mathbf{M} = \lambda\vec{v}$：

$$[v_1 \quad v_2]\begin{bmatrix} 6 & 4 \\ 2 & 4 \end{bmatrix} = \lambda[v_1 \quad v_2]$$

该式等价于线性系统

$$6v_1 + 2v_2 = \lambda v_1 \tag{2.4}$$

$$4v_1 + 4v_2 = \lambda v_2 \tag{2.5}$$

它可以表示为如下的齐次系统：

$$(\lambda - 6)v_1 - 2v_2 = 0$$

$$-4v_1 + (\lambda - 4)v_2 = 0$$

当且仅当该线性系统有一个非零解时，该系统的系数矩阵的行列式为零。因此

$$\begin{vmatrix} \lambda - 6 & -2 \\ -4 & \lambda - 4 \end{vmatrix} = \lambda^2 - 10\lambda + 16 = (\lambda - 8)(\lambda - 2) = 0$$

所以 M 的特征值为 $\lambda_1 = 8$ 和 $\lambda_2 = 2$ 。

如果将 $\lambda = 8$ 代入方程 2.4，可得

$$2v_1 - 2v_2 = 0$$

$$-4v_1 + 4v_2 = 0$$

从中可求得特征值 $\lambda = 8$ 的一个特征向量 $\vec{v} = [1 \quad 1]$。类似地，将 $\lambda = 2$ 代入，可得

$$-4v_1 - 2v_2 = 0$$

$$-4v_1 - 2v_2 = 0$$

从中可求得特征值 $\lambda = 2$ 的一个特征向量 $\vec{v} = [1 \quad -2]$。注意，特征向量的任何数乘都是与其特征值对应的特征向量。

上述结论可以自然地扩展到 $n \times n$ 序列，此时其特征方程是一个 n 次多项式。对于任何多项式，它的解的可能状态包括：无解，一个，或多达 n 个实根。应该意识到，对于 $n > 4$ 的情形，没有一般的闭形解存在。幸运的是，在计算机图形学中，我们一般只处理 4×4 矩阵或更小的矩阵。

2.9　欧几里得空间

从计算机图形学的观点来看，线性空间的一种名为欧几里得空间（Euclidean space）的特殊子集是最重要的。我们在前面对线性空间的讨论并没有提及“长度”和“正交性”，我们仅仅关注了向量空间。

2.9.1　内积空间

我们先从一个定义开始：设 $\mathcal{V}$ 是 $\mathbb{R}^n$ 上的向量空间。设 $\mathcal{V}^2$ 表示所有向量对 $(\vec{u}, \vec{v})$ 的集合，其中 $\vec{u}, \vec{v} \in \mathcal{V}$。内积（inner product）是 $\mathcal{V}^2$ 从 $\mathbb{R}^n$ 到的函数，记为 $\langle \vec{u}, \vec{v} \rangle$，它满足如下条件。

i．分配性：$\langle a_1\vec{u}_1 + a_2\vec{u}_2, \vec{v} \rangle = a_1\langle \vec{u}_1, \vec{v} \rangle + a_2\langle \vec{u}_2, \vec{v} \rangle$。

ii．交换性：$\langle \vec{u}, \vec{v} \rangle = \langle \vec{v}, \vec{u} \rangle$。

iii．正定性：$\langle \vec{u}, \vec{u} \rangle \geqslant 0$ 且 $\langle \vec{u}, \vec{u} \rangle = 0 \Longleftrightarrow \vec{u} = \vec{0}$。

空间 $\mathcal{V}$ 就是一个内积空间。正如我们在前面所提到的，向量空间一般可能具有任意类

型的元素，但由于我们仅关心实数多元组，因此在以下的相关讨论中，我们都是针对实数多元组而言的。

范数、长度和距离

显然存在许多 $\mathbb{R}^n$ 上的内积，即可以在这样的多元组上指定满足上述条件的任何的任意函数。我们在 2.3.4 节中介绍的点积是内积的一种特殊选择，它所具有的一些性质使它特别有用。内积定义的第三个条件涉及长度的定义，从中可知，任何非零向量都具有一个正值作为与其相关的内积。该内积的平方根叫做范数（norm），记为

$$\|\vec{u}\| = \sqrt{\langle \vec{u}, \vec{u} \rangle}$$

稍后我们将看到，欧几里得空间的几何“解释”允许我们把范数看成向量的长度。

如果向量的范数为 1，即 $\|\vec{u}\| = 1$，那么，我们就说该向量是规整化的（normalized）。任何（非零）向量 $\vec{u} \in \mathcal{V}$ 都能通过与 $1/\|\vec{u}\|$ 相乘而被正规化。两个向量 $\vec{u}, \vec{v} \in \mathcal{V}$ 之间的距离（distance）定义为 $\|\vec{v} - \vec{u}\|$。内积为点积的 $\mathbb{R}^n$ 上的内积空间定义为欧几里得空间。

2.9.2 正交和标准正交集

给定一个欧几里得空间 $\mathcal{V}$，等于 0 的内积特别重要：如果 $\langle \vec{u}, \vec{v} \rangle = 0$，则称它们是正交的（orthogonal）。

相对于基底向量的概念，正交性具有特别重要的地位。设 $\mathbf{V} = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 是向量空间 $\vec{v}$ 的基底向量集合。如果有 $\langle \vec{v}_i, \vec{v}_k \rangle = 0, \forall \vec{v}_i, \vec{v}_k \in \mathbf{V}, i \neq k$，则集合 $\mathbf{V}$ 叫做正交集（orthogonal）。

如果 $\mathbf{V}$ 是基底向量的正交集，并且 $\|\vec{v}_i\| = 1, \forall \vec{v}_i \in \mathbf{V}$，则进一步将 $\mathbf{V}$ 定义为是标准正交的（orthonormal）。具有标准正交坐标系的欧几里得空间称为笛卡尔空间（Cartesian space）。

任何欧几里得空间都有无数的定义空间的基底向量集合。任何基底向量集合都可能是正交的、标准正交的或非正交的。然而，任何基底向量集合都可以通过所谓的格拉姆－施密特正交化方法（Gram-Schmidt orthogonalization process）来转化成标准正交集。

在开始讨论正交化方法之前，我们必须先理解基底向量的标准正交集的一个性质：由于向量 $\mathbf{V}' = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_k\}$（其中 $k < n$，n 为 $\mathcal{V}$ 的维数）的一个标准正交集是某些基底向量集合的一个子集，因此它必定是线性无关的，而且，对任何 $\vec{u} \in \mathcal{V}$，向量

$$\vec{w} = \vec{u} - \langle \vec{u}, \vec{v}_1 \rangle \vec{v}_1 - \langle \vec{u}, \vec{v}_2 \rangle \vec{v}_2 - \cdots - \langle \vec{u}, \vec{v}_k \rangle \vec{v}_k \tag{2.6}$$

对每一个 $\vec{v}_i \in \mathbf{V}'$ 都是正交的。

设 $\mathcal{V}$ 是一个内积空间，并且 $\mathbf{V} = \{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 是它的一个基底。可以利用格拉姆－施密特正交化方法来构造一个正交基底 $\mathbf{U} = \{\vec{u}_1, \vec{u}_2, \ldots, \vec{u}_n\}$：

第 1 步　设 $\vec{u}_1 = \frac{\vec{v}_1}{\|\vec{v}_1\|}$。注意这一步使 $\vec{u}_1$ 具有单位长度。

第 2 步　设 $\vec{u}_2 = \frac{\vec{v}_2 - \langle \vec{v}_2, \vec{u}_1 \rangle \vec{u}_1}{\|\vec{v}_2 - \langle \vec{v}_2, \vec{u}_1 \rangle \vec{u}_1\|}$。注意 $\vec{u}_2$ 具有单位长度，并且对 $\vec{u}_1$ 是正交的（参见式（2.6））。这一步使集合 $\{\vec{u}_1, \vec{u}_2\}$ 标准正交化。

第 3 步　设 $\vec{u}_3 = \frac{\vec{v}_3 - \langle \vec{v}_3, \vec{u}_1 \rangle \vec{u}_1 - \langle \vec{v}_3, \vec{u}_2 \rangle \vec{u}_2}{\|\vec{v}_3 - \langle \vec{v}_3, \vec{u}_1 \rangle \vec{u}_1 - \langle \vec{v}_3, \vec{u}_2 \rangle \vec{u}_2\|}$。同样地，$\vec{u}_3$ 也具有单位长度，并且对 $\vec{u}_1$ 和 $\vec{u}_2$ 是正交的（参见式（2.6））。因此，集合 $\{\vec{u}_1, \vec{u}_2, \vec{u}_3\}$ 是标准正交化的。

第 4 步　重复上述步骤计算其余的 $\vec{u}_i$。

最后，我们得到 $\mathcal{V}$ 的一个标准正交基底。

2.10 最小二乘法

聪明的读者可能已经注意到，到目前为止，我们涉及的线性系统的方程的个数都恰好与未知数的个数完全相同。因此，只要存在一个惟一解，我们就能用多种方法来求解它。但是，在许多情形中，经常出现方程个数多于未知数的现象。

考虑一个简单的例子：如果存在两个点，我们希望确定通过这两点的直线的方程。可以建立一个线性系统来解决该问题。只要这两个点不重合，该系统具有两个方程和两个未知数，就可以计算出一个解。例如，假定有如下两点：

$$P_1 = (p_{1,1} \quad p_{1,2})$$

$$P_2 = (p_{2,1} \quad p_{2,2})$$

当然，它们定义了一条直线，可以表示为 $x_2 = mx_1 + b$。把点表示成线性系统并利用克莱姆定理可以求解系数 m 和 b。这两点必定在线上，因此满足直线方程，所以，可以把线性系统表示为

$$p_{1,1}m + b = p_{1,2}$$

$$p_{2,1}m + b = p_{2,2}$$

其矩阵形式为

$$\begin{bmatrix} p_{1,1} & 1 \\ p_{2,1} & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} p_{1,2} \\ p_{2,2} \end{bmatrix}$$

用克莱姆定理所求得的解为

$$m = \frac{\begin{vmatrix} p_{1,2} & 1 \\ p_{2,2} & 1 \end{vmatrix}}{\begin{vmatrix} p_{1,1} & 1 \\ p_{2,1} & 1 \end{vmatrix}} = \frac{p_{1,2} - p_{2,2}}{p_{1,1} - p_{2,1}}$$

$$b = \frac{\begin{vmatrix} p_{1,1} & p_{1,2} \\ p_{2,1} & p_{2,2} \end{vmatrix}}{\begin{vmatrix} p_{1,1} & 1 \\ p_{2,1} & 1 \end{vmatrix}} = \frac{p_{1,1}p_{2,2} - p_{2,1}p_{1,2}}{p_{1,1} - p_{2,1}}$$

然而，现在考虑有多于两个的点要拟合直线的情形。平面上任何三个点一般并不总在一条直线上。在这种情形下，我们希望得到一条与每个点的（垂直）距离最小的直线。这条直线可被看成是一个对每个给定的 x 值都返回一个 y 值的函数 f（确切地说，就是 $f(x) = mx + b$）。如图 2.9 所示。

我们可将该例表示为如下的线性系统：

$$p_{1,1}m + b = p_{1,2}$$

$$p_{2,1}m + b = p_{2,2}$$

$$p_{3,1}m + b = p_{3,2}$$

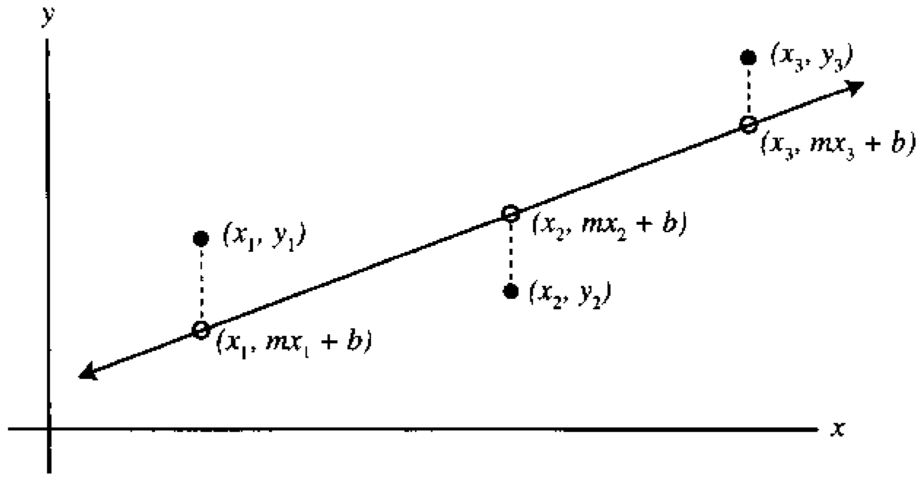


图 2.9　最小二乘法示例

其矩阵形式为

$$\begin{bmatrix} p_{1,1} & 1 \\ p_{2,1} & 1 \\ p_{3,1} & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} p_{1,2} \\ p_{2,2} \\ p_{3,2} \end{bmatrix}$$

注意，我们有三个方程，但只有两个未知数。这是一个所谓的超定系统（overdetermined system）。对于这类系统，我们希望得到的最好的结果是能满足一些目标条件的一些近似解。

点 (x_1, y_1) 与该直线之间的垂直距离 D_1 为 $|f(x_1) - y_1|$。基于许多不同的原因，我们需要确切地考虑这些点和该直线之间距离的平方和：

$$D^2 = (f(x_1) - y_1)^2 + (f(x_2) - y_2)^2 + \cdots + (f(x_n) - y_n)^2$$

所以，我们需要选取函数（直线）f，从而使 D^2 具有最小值。即我们需要选取 m 和 b 的近似值，使目标条件（D^2 具有最小值）能得以满足。这就是这种方法叫做最小二乘法（least squares）的原因。

为了理解这种方法是如何进行的，最好先让头脑休息一下。假设一个 n 维空间 $\mathcal{R}^n$，其中 n 为需要拟合（系统中方程）的点的数量。我们可以认为这个 n 维空间中位置 y 的坐标成员包含我们将要拟合的点的 y 值，即 $y = (y_1, y_2, \cdots, y_n)$。这个 n 维空间中的另一位置 $T(f)$ 可认为是具有包含一些点的 y 值，这些点位于我们用每一个点来拟合所得的拟合直线上，即 $T(f) = (f(x_1), f(x_2), \cdots, f(x_n))$。因此这些点与拟合的直线之间距离的平方和 D^2 就是（n 维空间 $\mathcal{R}^n$ 中的）向量 $T(f) - y$ 的距离的平方。

对于我们的例子中拟合直线到三点的情形，变换 T 对应于一个矩阵

$$\mathbf{M} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \end{bmatrix}$$

我们必须找到 f（即 m 和 b）使得 D^2（即 $\|T(f) - y\|$）具有最小值。将其表示为矩阵形式，可得

$$\mathbf{M}[f]-[y]=\mathbf{M}\begin{bmatrix} b \\ m \end{bmatrix}-[y]$$

$$=\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \end{bmatrix}\begin{bmatrix} b \\ m \end{bmatrix}-[y]$$

$$=\begin{bmatrix} b+mx_1-y_1 \\ b+mx_2-y_2 \\ b+mx_3-y_3 \end{bmatrix}$$

回忆一下，我们需要使 D^2 具有最小值，在前面我们已将它的一般形式表示为

$$D^2=\left(f(x_1)-y_1\right)^2+\left(f(x_2)-y_2\right)^2+\cdots+\left(f(x_n)-y_n\right)^2$$

针对现在的情形，有

$$D^2=\left(b+mx_1-y_1\right)^2+\left(b+mx_2-y_2\right)^2+\left(b+mx_3-y_n\right)^2$$

这就是一个函数，我们知道，当 D^2 对 b 和 m 的偏导数同时为零时，它将出现最小值。这样就得到需要求解的系统：

$$\left(b+mx_1-y_1\right)+\left(b+mx_2-y_2\right)+\left(b+mx_3-y_3\right)=0$$

$$x_1\left(b+mx_1-y_1\right)+x_2\left(b+mx_2-y_2\right)+x_3\left(b+mx_3-y_3\right)=0$$

如果用矩阵形式来表示，可得

$$\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \end{bmatrix}\begin{bmatrix} b+mx_1-y_1 \\ b+mx_2-y_2 \\ b+mx_3-y_3 \end{bmatrix}$$

为清晰起见，我们可将其改写为

$$\begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \end{bmatrix}\left(\begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \end{bmatrix}\begin{bmatrix} b \\ m \end{bmatrix}-\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}\right)=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

在这种形式中，我们可以发现矩阵 **M** 的不同组合形式：

$$\mathbf{M}^{\mathrm{T}}\left(\mathbf{M}\begin{bmatrix} b \\ m \end{bmatrix}-\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}\right)=\begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$$\mathbf{M}^{\mathrm{T}}(\mathbf{M}[f]-[y])=\mathbf{0}$$

$$\mathbf{M}^{\mathrm{T}}\mathbf{M}[f]-\mathbf{M}^{\mathrm{T}}[y]=\mathbf{0}$$

重新组织一下，得到

$$\mathbf{M}^{\mathrm{T}}\mathbf{M}[f]=\mathbf{M}^{\mathrm{T}}[y]$$

$$[f]=\left(\mathbf{M}^{\mathrm{T}}\mathbf{M}\right)^{-1}\mathbf{M}^{\mathrm{T}}[y]$$

聪明的读者可能已经发现，孤立 $[f]$ 的最后一步操作涉及一个可能不合理的运算，确

切地说，我们假定矩阵 M^TM 是可逆的。它的确是一个方阵，而且 x_1, x_2, x_3 是不相等的，所以该矩阵是可逆的。

2.11 推荐的阅读材料

关于线性代数的书籍浩如烟海，在一个网上书店站点中对字符串“线性代数”的最近一次搜索产生了 465 条结果。特别合适的书籍是线性代数的大学课本，比如：

Jeanne Agnew 和 Robert C. Knapp, *Linear Algebra with Applications*, Brooks/Cole, Monterey, CA, 1978.

Howard Anton, *Elementary Linear Algebra*, John Wiley and Sons, New York, 2000.

非常有用且易于理解的还有：

Seymour Lipschutz, Schaum's Outline of Theory and Problems of Linear Algebra, McGraw-Hill, New York, 1968.

在计算机图形学领域，下列的书籍包含许多与线性代数有关的内容：

M. E. Mortenson, *Mathematics for Computer Graphics Applications*, Industrial Press, New York, 1999（1－3 章）.

James D. Foley, Andries van Dam, Steven K. Feiner 和 John F. Hughes, *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley, Reading, MA, 1996（附录：用于计算机图形学的数学）.

Gerald Farin 和 Dianne Hansford, *The Geometry Toolbox for Graphics and Modeling*, A. K. Peters, Natick, MA, 1998.

第 3 章　向量代数

3.1　向量基础

在讨论向量空间之前，我们先回顾一下几个基本概念。我们讨论一下物理学问题以使问题显得具体一些。可用一个单独的数值来表示物理性质和现象的基础类型，例如，质量、体积、距离或温度。但是，性质和现象的二级类型就不能这样表示，它们是多值的（multiple-valued）。这种类型的一个例子是物体的运动，它包含两个方面，即速度和方向（比如说，千米每小时和航向罗盘方位）。性质的第一种类型可称为数量值的，第二种类型可称为向量值的。向量值的实体也可叫做多维的（multidimensional），构成实体的各个值叫做它的分量（component）。

稍后我们将讨论一些关于这些向量值数的数学概念，但现在还是先来看一种特殊的由距离和方向所定义的子集，我们把它们叫做向量。为了形象地表示这类实体，即向量，我们用箭头（有方向的线段）来表示它们，如图 3.1 所示。方向由箭头的指向来表示，而距离由其长度来表示。

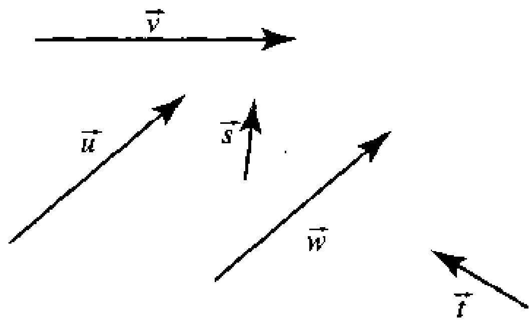


图 3.1　向量表示为有向线段

3.1.1　向量等价

向量的最重要的特性之一是等价性（equivalence），其含义就是两个向量相等与否。向量的分量包括其方向和长度，而“位置”并非其定义的一部分。在图 3.1 中，向量 $\vec{u}$ 和 $\vec{t}$ 具有不同的方向和长度，因此是不同的向量。但向量 $\vec{u}$ 和 $\vec{w}$ 具有相同的方向和长度，因而是等价的，即使它们被画在图中的不同位置上也是如此。如果这显得不太直观，可以按如下的方式来理解：你从你的房子出发或从你的邻居的房子出发，向东行走 3 千米，所走的相对路径是相等的。

3.1.2　向量加法

正确地理解了向量的方向和距离的含义后，现在来讨论向量的运算。首先讨论加法，即把两个或更多向量相加有什么意义。回到我们说过的关于行走的直观例子，加法对应的是沿一定的方向行走一定的距离（一个向量），然后马上沿另一方向行走一定的距离。这样，你将到达你的终点，这将对应于另一个向量，它可定义为前两个向量之和。

例如，如图 3.2 所示，有两个向量$\vec{u}$ 和$\vec{v}$，表示两段行程。记住，向量的位置是不重要的，因此，可以像图 3.3 那样地重画它们。从行程的起点到终点的更直接的路径是两个向量之和，可以画成“首尾相接”，这将更明确地说明它们之和是向量$\vec{w}$，如图 3.4 所示。这种操作可用数学方法表示为 $\vec{u}+\vec{v}=\vec{w}$。

由于两个向量相加总是得到另一个新向量，因此我们可以把这种思想扩展到向量加法“链”，如图 3.5 所示，其中表示了向量和 $\vec{s}+\vec{t}+\vec{u}+\vec{v}=\vec{w}$。

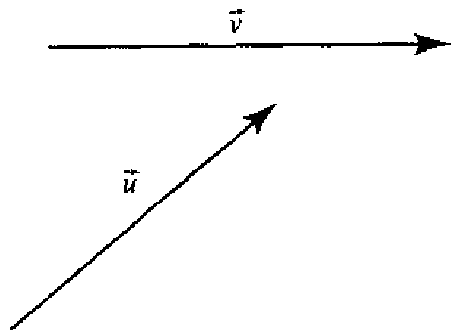


图 3.2　两个向量

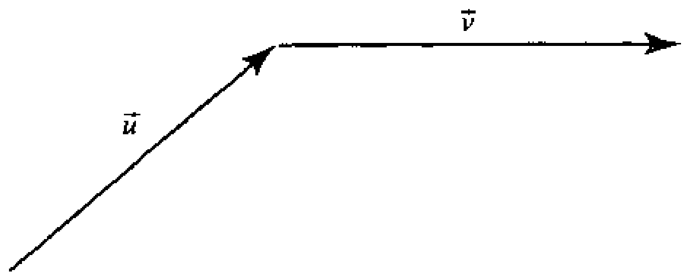


图 3.3　首尾相接的两个向量

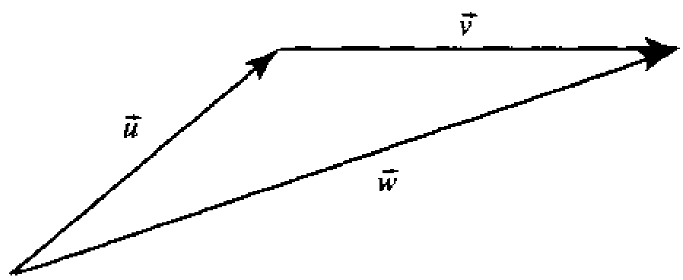


图 3.4　向量加法

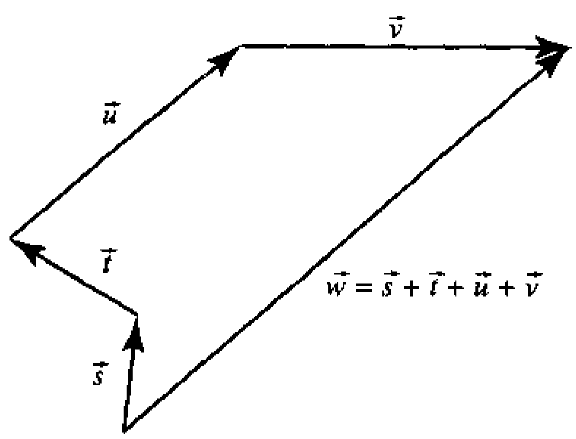


图 3.5　向量加法链

3.1.3　向量减法

我们也可以从一个向量中减去另一个向量。在直观意义上，这等价于沿一定方向行走特定的距离，然后再沿一定方向的反方向行走另一距离。如图 3.6 所示。

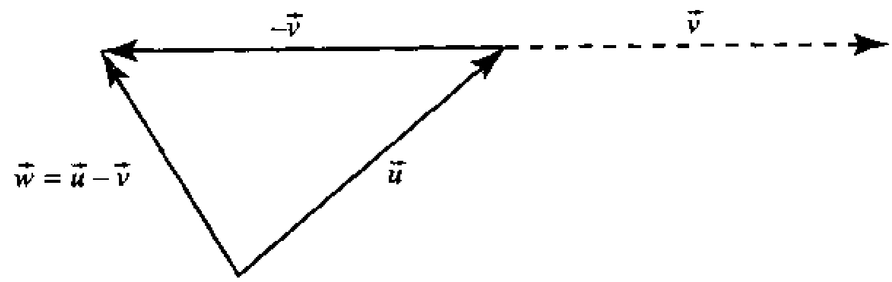


图 3.6　向量减法

3.1.4　向量数乘

对向量的另一种运算是数量乘法（scalar multiplication）或缩放（scaling）。其意义就是改变向量的长度而不改变其方向。在图 3.7 中可以看到，向量 $\vec{v}$ 的方向与 $\vec{u}$ 的方向完全相同，但前者的长度是后者的两倍。在数学上可表示为 $\vec{v}=2\vec{u}$。

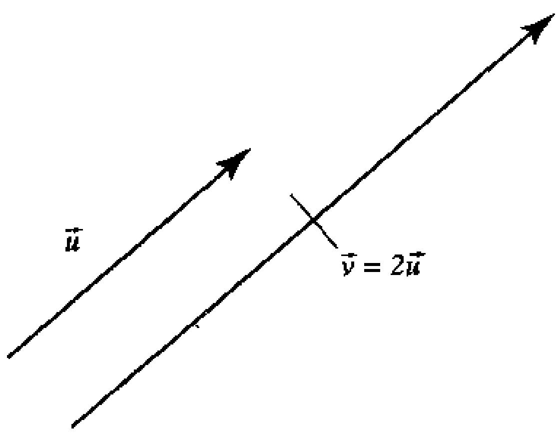


图 3.7　向量数乘

一般地，数乘就是$\vec{v}=\alpha\vec{u}$，其中α是任意实数。在上例中，我们有$\alpha=2$，但我们可以简单地将α设为负值。图 3.8 显示了$\vec{u}$被$\alpha=-1$所缩放的结果（非常明显，这可以称为向量反转（vector negation））。

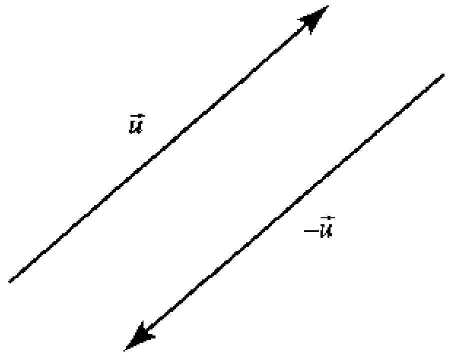


图 3.8　向量反转

3.1.5　向量加法和数乘的性质

至此，我们已经定义了向量的加法和数乘。你可能想知道，组合这两种运算的规则是否与已知的简单数（即实数）的普通加法和乘法的规则相同。

我们来考察一下几个常用的性质。

i．交换性：$\vec{u}+\vec{v}=\vec{v}+\vec{u}$。从图 3.9 中可以看出，不管各项的次序如何，和是相同的。

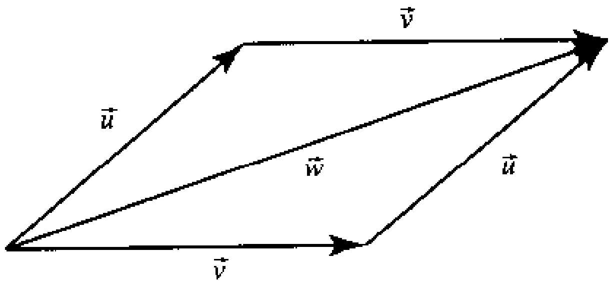


图 3.9　向量加法的交换性

ii．结合性：$\vec{u}+(\vec{v}+\vec{w})=(\vec{u}+\vec{v})+\vec{w}$。从图 3.10 中可以看出，不管各项如何分组，和是相同的。

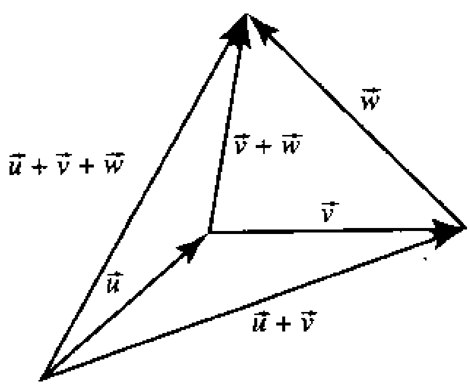


图 3.10　向量加法的结合性

iii. 乘法对加法的分配性：$(\alpha+\beta)\vec{u}=\alpha\vec{u}+\beta\vec{v}$。如图 3.11 所示。

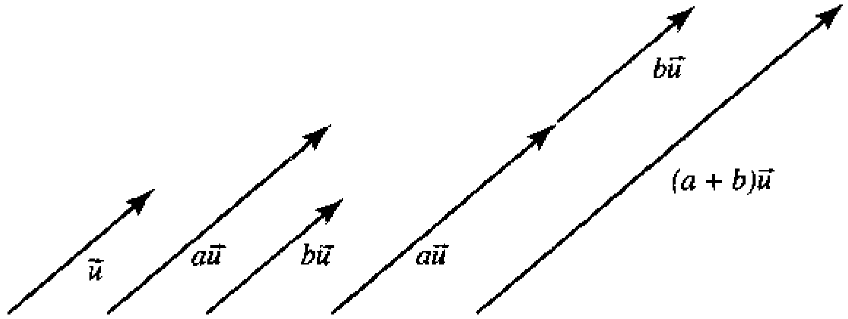


图 3.11　乘法对加法的分配性

iv. 加法对乘法的分配性：$\alpha(\vec{u}+\vec{v})=\alpha\vec{u}+\alpha\vec{v}$。从图 3.12 中可以看出，不管各项如何分组，和是相同的。

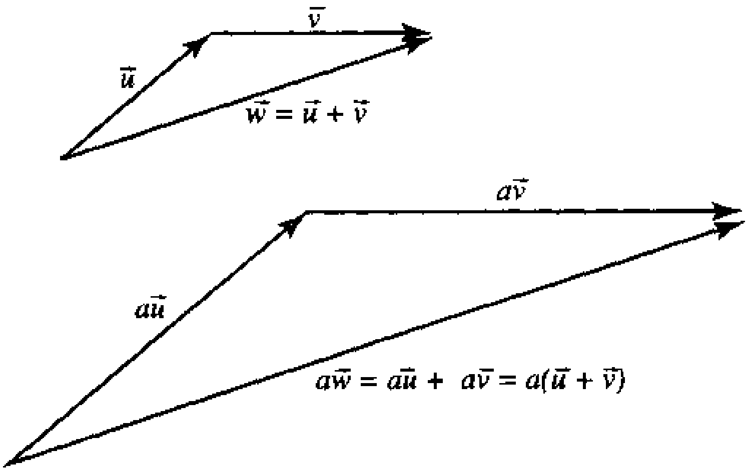


图 3.12　加法对乘法的分配性

3.2　向量空间

我们一直把向量说成是“有向线段”，向量空间（vector space）这一抽象概念可定义为由一组集合 $\mathcal{V}$ 构成的（实数上的）向量空间集合 $\mathcal{V}$ 的元素叫做“向量”。向量空间它具有如下性质。

i. 定义了向量的加法（或减法），其结果是另一个向量。

ii. 集合 $\mathcal{V}$ 对线性组合是封闭的：如果有 $\vec{u},\vec{v}\in\mathcal{V}$，且 $\alpha,\beta\in\mathbb{R}$，则 $\alpha\vec{u}+\beta\vec{v}\in\mathcal{V}$。

iii. 存在一个惟一的向量 $\vec{0}\in\mathcal{V}$，叫做零向量（zero vector），它具有以下性质：

$$\text{a. } \forall\vec{v}\in\mathcal{V},0\cdot\vec{v}=\vec{0},\text{ 其中 } 0\in\mathbb{R}$$
$$\text{b. } \forall\vec{v}\in\mathcal{V},\vec{0}+\vec{v}=\vec{v}$$

注意，除了我们还未讨论向量与向量的乘法之外，所有这些性质都直观地与向量定义的“有向线段”版本相吻合。还应注意，“对线性组合是封闭的”包括了我们已讨论过的向量与数量的乘法。

3.2.1　生成空间

给定一个向量集合$\{\vec{v}_1,\vec{v}_2,\cdots,\vec{v}_n\}\in\mathcal{V}$，这些向量的所有线性组合的集合$S$是一个包含另一个向量空间的向量的（无穷）集合，该空间叫做由$\{\vec{v}_1,\vec{v}_2,\cdots,\vec{v}_n\}$所生成的空间。即任何向量$\vec{w}\in S$都可以表示成$\vec{w}=\lambda_1\vec{v}_1+\lambda_2\vec{v}_2+\cdots+\lambda_n\vec{v}_n$，其中$\lambda_i\in\mathbb{R}$。集合$\{\vec{v}_1,\vec{v}_2,\cdots,\vec{v}_n\}$叫做$S$的生成集（spanning set）。

这里有一个例子，可以更清晰地解释这一概念：如果有两个（不平行的）向量$\vec{u}$和$\vec{v}$，它们是三维空间的有向线段，那么，这两个向量生成的空间包含$\vec{u}$和$\vec{v}$所定义的空间内的所有向量（如图3.13所示）。

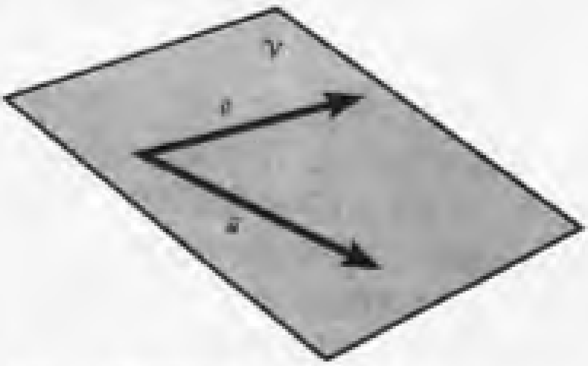


图3.13　三维空间中两个向量的生成空间是一个平面

3.2.2　线性无关

假定有向量集合$\vec{u}$和$\vec{v}$，它们生成空间S。注意在图3.13所示的例子中，我们规定向量$\vec{u}$和$\vec{v}$必须是不平行的。直观地看，如果它们是平行的，我们将不能用它们来定义一个平面，而只能定义一条直线。考虑下面的情形，有三个向量$\vec{u}$，$\vec{v}$和$\vec{w}$，并且$\vec{w}=\alpha\vec{u}$。我们仍然定义一个平面，而且这三个向量将生成同样的集合S。所以，$\vec{u}$或$\vec{w}$可认为是多余的。

这种直觉观点可形式化为线性无关的定义：集合$\{\vec{v}_1,\vec{v}_2,\cdots,\vec{v}_n\}\in\mathcal{V}$是线性相关的，如果存在不全为零的数量$\lambda_1,\lambda_2,\cdots,\lambda_n$，使得

$$\lambda_1\vec{v}_1+\lambda_2\vec{v}_2+\cdots+\lambda_n\vec{v}_n=\vec{0}$$

如果仅当$\lambda_1=0,\lambda_2=0,\cdots,\lambda_n=0$时有

$$\lambda_1\vec{v}_1+\lambda_2\vec{v}_2+\cdots+\lambda_n\vec{v}_n=\vec{0}$$

则称它们是线性无关的。

更直观地说，上述定义意味着，当且仅当不存在$\vec{v}_i$是集合中其他向量的线性组合时，向量集合是线性无关的。

3.2.3　基底、子空间和维数

如果有向量空间S，那么如果满足

i．$\{\vec{v}_1,\vec{v}_2,\cdots,\vec{v}_n\}$是线性无关的

ii．$\{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n\}$是$S$的一个生成集

则集合$\{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n\}$是$S$的一个基底。

如果有向量空间$\mathcal{V}$和一些基底向量集合$V = \{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n\} \in \mathcal{V}$，那么，$V$生成的空间$S$叫做$\mathcal{V}$的子空间（subspace）。$S$的维数$n$定义为$S$中线性无关向量的最大个数。

为了说明得更具体一些，看一下图 3.13 所示的例子。其中有一个向量空间，它由三维空间（即$\mathcal{V} = \mathbb{R}^3$）中的所有有向线段组成，其基底向量就是图示的两条有向线段（即$V = \{\vec{u}, \vec{v}\}$），由V生成的空间S就是这两个向量所在的平面。S的维数为2。

注意如下要点，$\mathcal{V}$的任何子空间S都有无数的生成集。回到前面的例子，平面中的任何两条不平行的有向线段都构建一个三维空间的平面子集的基底。

假定有向量集合$V = \{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n\} \in \mathcal{V}$，它是线性无关的。$V$的生成空间中的任何其他向量$\vec{w}$可描述为$V$的一个线性组合：

$$\vec{w} = x_1\vec{v}_1 + x_2\vec{v}_2 + \cdots + x_n\vec{v}_n, \qquad x_i \in \mathbb{R}$$

注意，对于给定的$\vec{w}$，要素x_i是惟一的（否则向量$\vec{v}_i$是线性无关的）。因此，我们可以定义向量集合$V=\{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n\} \in \mathcal{V}$，它是线性无关的，并构成$\mathcal{V}$的一个基底。我们把元素$x_i\vec{v}_i$叫做$\vec{w}$的分量（component），系数$x_i$叫做$\vec{w}$的坐标（coordinate）。

举一个例子可以更清晰地说明上述概念。图 3.14 显示了与图 3.13 相同的两个向量$\vec{u}$和$\vec{v}$。因为它们是线性无关的（即不平行且都不是$\vec{0}$向量），所以它们构成一个基底。

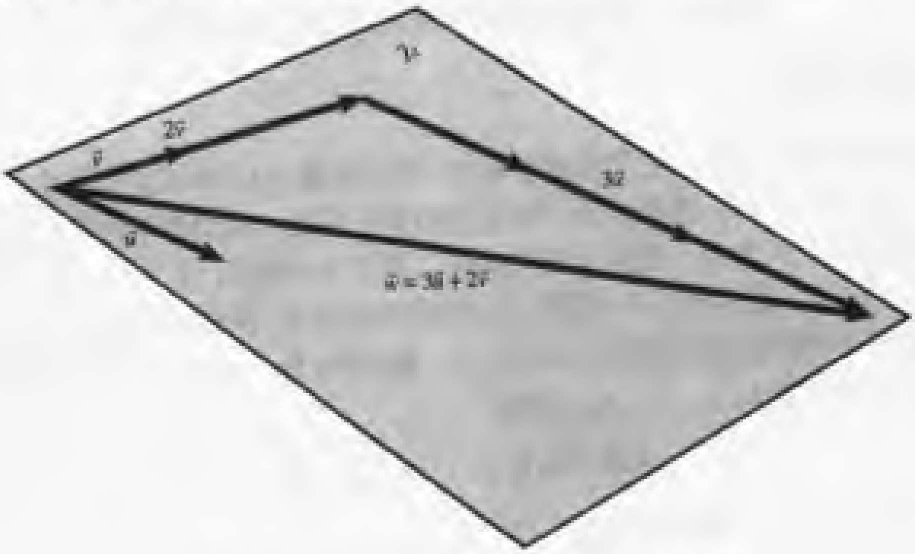


图 3.14　一个以向量作为基底向量的线性组合

$\vec{u}$和$\vec{v}$的生成空间$\mathcal{V}$中的向量$\vec{w}$可描述为基底向量的一个线性组合（具体地说，$\vec{w} = 3\vec{u} + 2\vec{v}$）。可见，在直观上，$\vec{w}$的系数（坐标）只能是$x_1 = 3, x_2 = 2$（用图来证明）：假定$x_1$不是3，注意，那将没有可能的$x_2$值以求得$\vec{w}$。

我们需要提供上述论断的正式证明，即需要证明如下较简单的命题：设$V = \{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n\}$是$\mathcal{V}$的一个基底，则$\sum c_i\vec{v}_i = 0 \Leftrightarrow c_1 = c_2 = \cdots = c_n = 0$。

【证明】我们须从两个方面来证明。我们先证明简单的方面：如果$c_1 = c_2 = \cdots = c_n = 0$，则根据加法和乘法的定义自然有$\sum c_i\vec{v}_i = 0$。为了证明另一个方面，假定$\sum c_i\vec{v}_i = 0$且对某

些 j 有 $c_j \neq 0$。那么

$$\vec{v}_j = \sum_{i \neq j} \frac{-c_i}{c_j} \vec{v}_i$$

这将与线性无关的假设矛盾，因此 $c_1 = c_2 = \cdots = c_n = 0$。■

上述命题可更直观地表述如下：零向量只能被描述为所有常数都为零的基底向量的线性组合，反之，如果线性组合的所有常数都为零，则它只能定义零向量。

现在我们来证明线性组合是惟一的。更正式的陈述如下：设 $V = \{\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n\}$ 是 $\mathcal{V}$ 的一个基底，则 $\mathcal{V}$ 中的每一个向量都可写成 $\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_n$ 的惟一的一个线性组合。

【证明】 假定有 $\vec{w} \in \mathcal{V}$，那么存在 $c_1, c_2, \ldots, c_n \in \mathbb{R}$，使得

$$\vec{w} = c_1\vec{v}_1 + c_2\vec{v}_2 + \cdots + c_n\vec{v}_n$$

假定这些常数不是惟一的，即

$$\vec{w} = d_1\vec{v}_1 + d_2\vec{v}_2 + \cdots + d_n\vec{v}_n$$

其中一些 $c_i \neq d_i$。但是，如果这是真的，那么

$$\vec{0} = (d_1 - c_1)\vec{v}_1 + (d_2 - c_2)\vec{v}_2 + \cdots + (d_n - c_n)\vec{v}_n$$

我们已知，前述的命题，即生成零向量 $\vec{0}$ 的线性组合的系数全都为零。这意味着 $d_i = c_i, \forall i \in 1 \ldots n$。这证明每一个 $\vec{w} \in \mathcal{V}$ 都可定义为基底向量的一个惟一的线性组合。■

3.2.4　方向

假定有两个线性无关向量 $\vec{u}$ 和 $\vec{v}$。我们知道，它们可以定义一个平面，如果这两个向量是三维的，则它们将定义三维空间的一个平面。图 3.15 显示了这两个向量 $\vec{u}$ 和 $\vec{v}$，以及它们之间的夹角 $\theta_{\vec{u}\vec{v}}$。注意，我们正在“开发”一些从未描述过的表示方法。我们根据词典顺序来说明向量（先 $\vec{u}$ 后 $\vec{v}$），并据此次序来说明它们之间的夹角（$\theta_{\vec{u}\vec{v}}$），图 3.15 中角度的箭头的方向是从 $\vec{u}$ 到 $\vec{v}$，这说明角度是正的。一般来说，角度的方向是“逆时针方向”，即角度沿该标准方向增大。但是，所有这些惯例都是基于一个假设，该假设非常明显，人们甚至可能不会想到它：所有这些都是画在具有一个内置的第三维的页上，该页也具有一个明确的第三个方向，即“从页面向页外”的方向（页有厚度，画画的墨水画在其中的一个表面上，我们把该表面作为“正面”，相对的另一面是“背面”，因此，逆时针方向是非常明确的）。

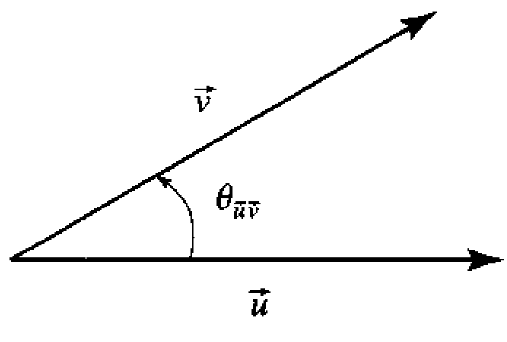


图 3.15　向量之间的夹角

在图 3.15 中，平面是无穷薄的平面，如果我们从另一面来看，会出现怎样的情况呢？"页内"或"页外"的观念将使逆时针（即方向）的含义相反。所以，我们实际上不能区分方向。我们只不过是被把图打印在页上的确定惯例所"欺骗"，特别是被"从页面向页外"和"从页外到页内"的观念所欺骗。最后的事实说明，正是第三个方向的观念使我们可以定义方向，并使我们能够摆脱不确定性。

现在，除了基底向量$\vec{u}$和$\vec{v}$，假定有第三个（线性无关）向量$\vec{w}$，我们可以认为$\vec{w}$的方向是"从页面向页外"。当然，相对于$\vec{u}$和$\vec{w}$，$\vec{v}$也可用来作为第三个向量，依此类推。这（最终）将允许我们把基底的方向或符号定义如下：

$$\operatorname{sgn}(\vec{u},\vec{v},\vec{w})=\operatorname{sgn}(\theta_{\vec{u}\vec{v}})$$

回到图 3.15，$\vec{w}$指向页内还是页外取决于如何选择我们的表示惯例。对应于关于一页纸的正面和背面的观念，选择$\vec{w}$指向页外似乎更"自然"一点。这种惯例称为"右手法则"，因为如果伸出右手，沿正向旋转方向弯曲手指，那么，定义的方向对应于拇指所指的方向，如图 3.16 所示。如果基底的符号是正的，我们使用如下的表示法

$$\operatorname{sgn}(\vec{u},\vec{v},\vec{w})=+1$$

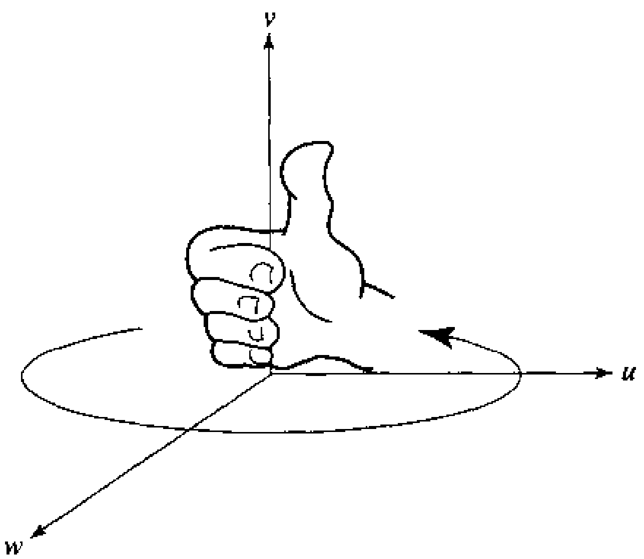


图 3.16　确定方向的右手法则

3.2.5　基底变化

我们已经说明，特定空间中的每一个向量都是基底向量的一个特定集合的惟一的线性组合。但是，这并不意味每一个向量都有惟一的线性组合表示。对任何给定的维数为 n 的向量空间$\mathcal{V}$，存在无数的线性无关的$\mathcal{V}$的 n 维子集。即，任何 $\vec{w}\in\mathcal{V}$ 都可以表示为任何任意选取的基底向量集合的一个线性组合。图 3.14 中的向量$\vec{w}=3\vec{u}+2\vec{v}$也可以表示为$\vec{w}=3\vec{s}+\vec{t}$，如图 3.17 所示。

在直观上这显然是成立的。对该结论成立的原因正式的解释非常有用：假定有两个$\mathcal{V}$的基底向量的集合$\vec{a}_1,\vec{a}_2,\ldots,\vec{a}_n$和$\vec{b}_1,\vec{b}_2,\ldots,\vec{b}_n$。我们知道，$\mathcal{V}$的任何向量都能用任何基底来表示，对于组成其他基底向量集合的向量，这当然也成立。即每一个$\vec{a}$都可以用$\vec{b}$组成

的基底来表示：

$$\vec{a}_k = c_{1,k}\vec{b}_1 + c_{2,k}\vec{b}_2 + \cdots + c_{n,k}\vec{b}_n \tag{3.1}$$

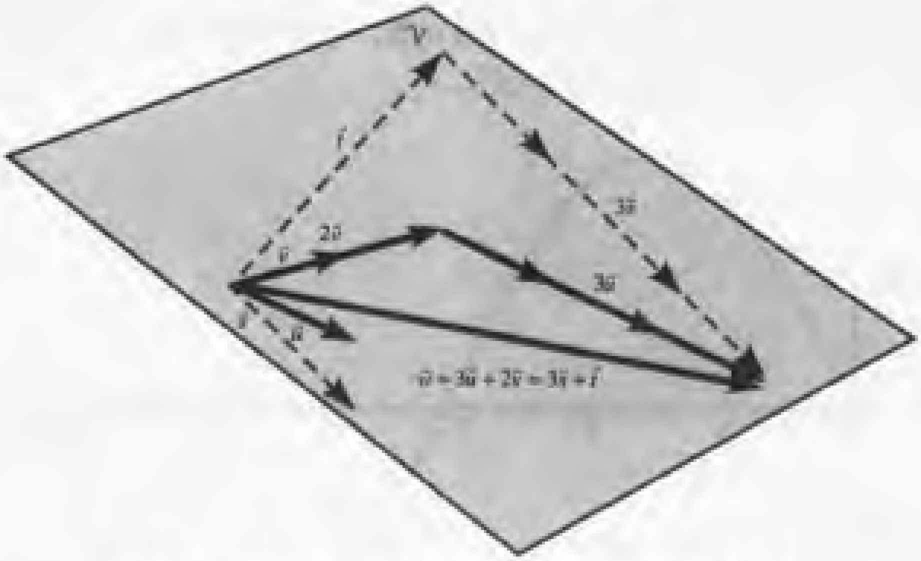


图 3.17　一个向量表示为两组不同的基底向量的线性组合

如果有一个向量 $\vec{w} \in \mathcal{V}$，它可表示为 $\vec{a}_1, \vec{a}_2, \ldots, \vec{a}_n$ 的一个线性组合，用该组合替代表达式（3.1）中的 $\vec{a}_k$，可得

$$\begin{aligned}
\vec{w} &= d_1\vec{a}_1 + d_2\vec{a}_2 + \cdots + d_n\vec{a}_n \\
&= d_1(c_{1,1}\vec{b}_1 + c_{2,1}\vec{b}_2 + \cdots + c_{n,1}\vec{b}_n) + \\
&\quad d_2(c_{1,2}\vec{b}_1 + c_{2,2}\vec{b}_2 + \cdots + c_{n,2}\vec{b}_n) + \\
&\quad \cdots + d_n(c_{1,n}\vec{b}_1 + c_{2,n}\vec{b}_2 + \cdots + c_{n,n}\vec{b}_n) \\
&= (d_1c_{1,1} + d_2c_{1,2} + \cdots + d_nc_{1,n})\vec{b}_1 + \\
&\quad (d_1c_{2,1} + d_2c_{2,2} + \cdots + d_nc_{2,n})\vec{b}_2 + \\
&\quad \cdots + (d_1c_{n,1} + d_2c_{n,2} + \cdots + d_nc_{n,n})\vec{b}_n
\end{aligned}$$

在下一章中我们将看到，上述繁琐的计算可利用矩阵乘法来简单地实现。

3.2.6　线性变换

在钻研这一部分内容之前，我们先来复习一些预备知识。如果有两个集合 $\mathcal{D}$ 和 $\mathcal{R}$，那么可以定义一种操作来将 $\mathcal{D}$ 中的每一个元素与 $\mathcal{R}$ 中的每一个元素严格地对应。如果对 $\mathcal{D}$ 中的所有元素进行这种操作，那么可以将操作结果视为元素对 (a, b) 的集合，其中 $a \in \mathcal{D}$，$b \in \mathcal{R}$。我们将这种对应两个集合的元素的操作称为函数（function）、变换（transformation）或映射（mapping）（它们是等价并可交换使用的术语）。集合 $\mathcal{D}$ 叫做定义域（domain），集合 $\mathcal{R}$ 叫做值域（range）。注意如下事实是很重要的，即定义域和值域都有可能是有限的，也有可能是无限的。而且，函数可能是连续的，也可能是非连续的。值域中的许多值可能

映射到值域中的相同值上，而且定义域中的每一个值可能映射到值域中的一个值上。

经常使用某些种类的图形来描述函数，定义域作为水平轴，值域作为垂直轴，而函数的值显示为直线或曲线，或者条形图。集合$\{(x, f(x)) : x \in \mathcal{D}\} \subset \mathcal{D} \times \mathcal{R}$被定义为函数的图形。一个规范的函数实例是三角函数中的正弦函数。它的定义域包含所有的实数；其值域是-1 到 1 之间的所有实数（如图 3.18 所示）。

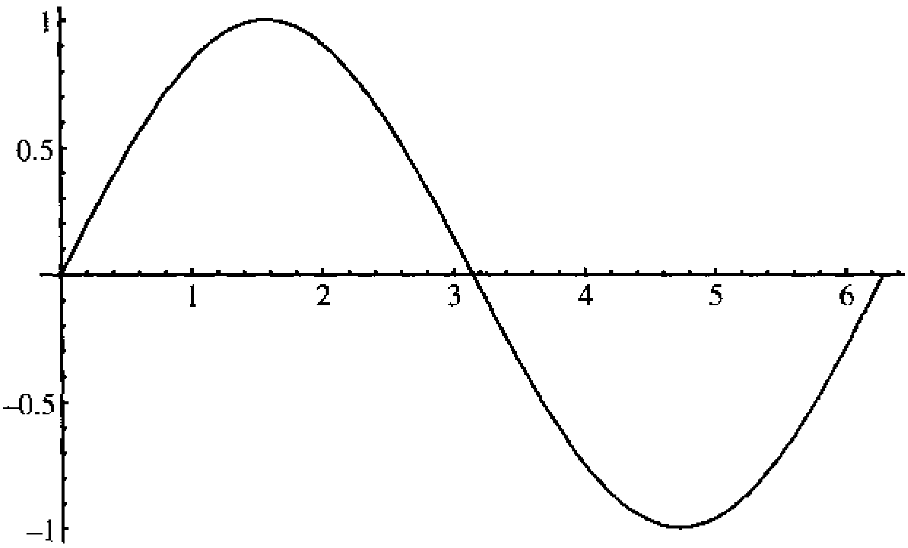


图 3.18 正弦函数

在这些讨论的内容中，我们感兴趣的的部分称为线性变换，即从一个线性（向量）空间映射到另一个线性空间的变换。一般地，两个向量空间之间的线性变换$\mathcal{U}$和$\mathcal{V}$是一个映射T：$\mathcal{U} \to \mathcal{V}$，使得

i．$T(\vec{u}+\vec{v}) = T(\vec{u}) + T(\vec{v})$ 对任何向量$\vec{u}, \vec{v} \in \mathcal{V}$ 都成立

ii．$T(\alpha\vec{u}) = \alpha T(\vec{u})$对任何$\alpha \in \mathbb{R}$ 向量 $\mathbb{R}$ 和任何 $\vec{u} \in \mathcal{V}$ 都成立

线性变换常常被提及的性质是保持线性组合。回忆一下，向量的线性组合定义为$x_1\vec{v}_1 + x_2\vec{v}_2 + \cdots + x_n\vec{v}_n, x_i \in \mathbb{R}$，可被分解为上述的两种运算，并且可以看出它们是等价的条件。线性变换总是把直线映射成直线（或者 0），并且总是满足$T(\vec{0}) = \vec{0}$。图 3.19 能够给你一些直观的概念。图中显示的是“放大两倍”的情形，从中可以直接看到线性组合被保持。

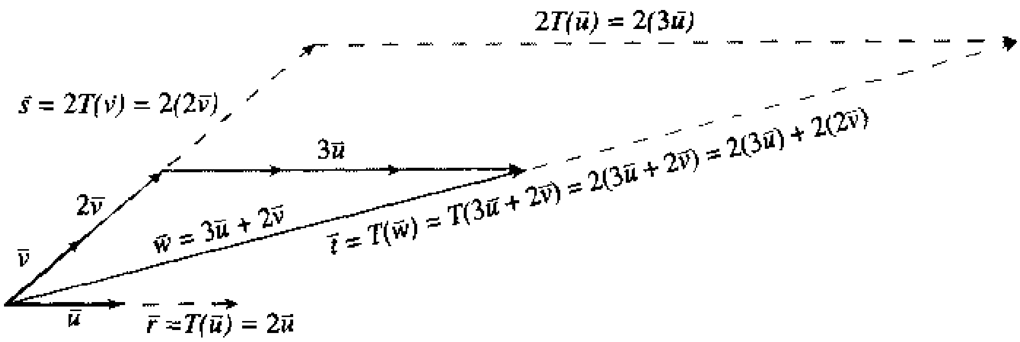


图 3.19 线性变换“放大两倍”

$$\vec{t} = T(\vec{w}) = 2(3\vec{u} + 2\vec{v}) = 2(3\vec{u}) + 2(2\vec{v})$$

由于线性变换保持线性组合，而且所有的向量 $\vec{u}_i \in \mathcal{V}$ 都可表示成一些基底向量 $\{\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n\}$ 的线性组合，因此线性变换的性质可通过它对基底向量的操作结果来描述。通过考虑对向量可执行怎样的变换，我们就能理解线性变换能执行的操作的类型：改变其长度（缩放）或方向（旋转）。图 3.19 中显示的是一种均衡缩放，即基底向量同时用相同的值来缩放；然而，当然也允许对每一个向量用不同的值来缩放，此时我们将得到非均衡缩放（图 3.20 显示了一个基底向量放大两倍、其余的基底向量放大 1.5 倍的结果）。如果用同样的方法旋转基底向量，那么将实现那些基底向量的线性组合的所有向量（相同的数量）的旋转（如图 3.21 所示）。最后，剪切变换只缩放基底向量中的一个（如图 3.22 所示）。

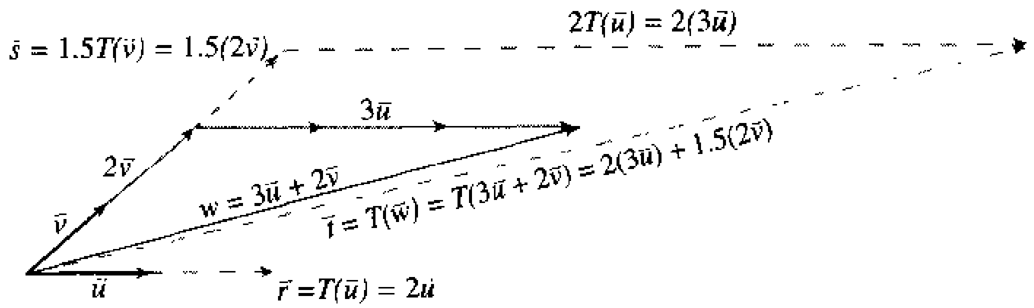


图 3.20　非均衡缩放线性变换

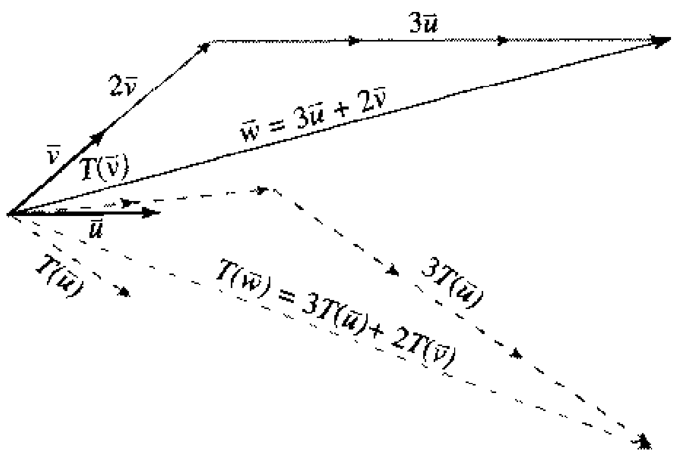


图 3.21　旋转变换

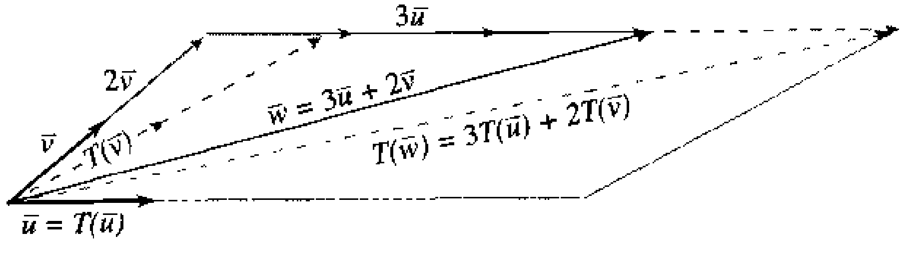


图 3.22　剪切变换

3.3 仿射空间

到目前为止，我们讨论的都是向量——什么是向量，它们能做什么，等等。但是，什么是点呢？世界可以被看成是点（位置）的空间。怎样才能把点与我们已经讨论过的向量联系起来呢？显然，如果给定一个点，我们就将一个向量“附加”到该点上，而且在该向量的终点还存在另一个点。其次，如果有两个点，则存在一个向量从其中一个点指向另一点，反之亦然。

因此，我们很容易从功能上来区分点和向量。为了清晰地说明它们的区别，我们采用一种通用的表示法：向量总是用上方附加箭头（$\vec{u},\vec{v}$）或“帽子”（$\hat{u},\hat{v}$）（表示单位向量）的小写字母来表示；点则用没有箭头的大写字母（P,Q）来表示。由于两点可以确定一个向量，因此有时我们使用一种表示法来明确地表示这一点，即用$\vec{pq}$来表示从P到Q的向量。

一个仿射空间（affine space）$\mathcal{A}$由一个点集$\mathcal{P}$和一个向量集$\mathcal{V}$所构成，它是由一些基底或$\mathcal{V}$的基所生成的向量空间。$\mathcal{A}$的维数n就是$\mathcal{V}$的维数。我们将$\mathcal{A}$中的点表示为$\mathcal{A.P}$，向量表示为$\mathcal{A.V}$。

上述定义直观地说明了点空间与构建一个仿射空间的向量空间之间的关系。这种关系可以更正规地用定义点对之间的减法的公理来确定，即所谓的“首尾相接”公理：

i．$\forall P,Q\in\mathcal{A.P},\exists$ 一个惟一的向量$\vec{v}\in\mathcal{A.V}$，使得$\vec{v}=P-Q$。

ii．$\forall Q\in\mathcal{A.P},\forall\vec{v}\in\mathcal{A.V},\exists$ 一个惟一的点P，使得$P-Q=\vec{v}$。

iii．$\forall P,Q,R\in\mathcal{A.P},(P-Q)+(Q-R)=P-R$。

注意，上面的条件（i）也可以表示成$P=Q+\vec{v}$，并包含$P=P+\vec{0}$。图 3.23 显示了前两个公理，图 3.24 显示了“首尾相接”公理。

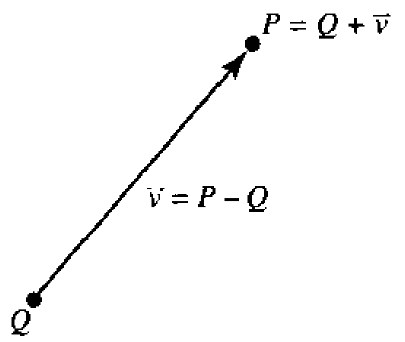


图 3.23　点的减法

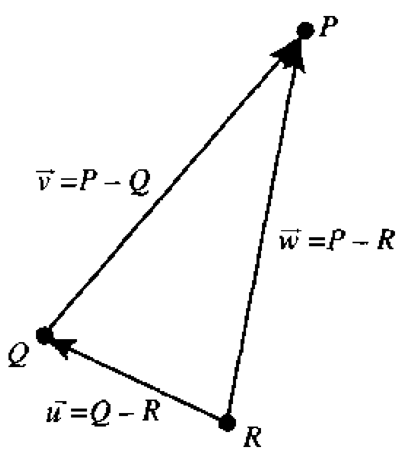


图 3.24　“首尾相接”公理

最后，还有一个公理，DeRose 将它叫做坐标公理。该公理定义了点的两种重要的乘法运算：

$$\forall P\in\mathcal{A.P},1\cdot P=P\text{ 且 }0\cdot P=\vec{0}$$

它简明地告诉我们一个点乘以 1 结果还是它自己，一个点乘以 0 结果是$\mathcal{A.V}$中的零向量。

下面是几个有用的等式（DeRose 1992）：

i. $Q-Q=\vec{0}$

【证明】设$Q=R$，则“首尾相接”公理可以重写为$(P-Q)+(Q-Q)=P-Q$，这意味着$(Q-Q)=\vec{0}$。■

ii. $R-Q=-(Q-R)$

【证明】设$P=R$，则“首尾相接”公理可以重写为$(R-Q)+(Q-R)=R-R$。由于$R-R=\vec{0}$，这就暗示$(R-Q)=-(Q-R)$。■

iii. $\vec{v}+(Q-R)=(Q+\vec{v})-R$

【证明】让$\vec{v}=P-Q$。将之代入“首尾相接”公理，得到$\vec{v}+(Q-R)=P-R$。再将$Q+\vec{v}$代入则可以得到结果。■

iv. $Q-(R+\vec{v})=(Q-R)-\vec{v}$

【证明】上面的等式iii乘以−1 即得。■

v. $P=Q+(P-Q)$

【证明】我们根据加法的定义，将“首尾相接”公理重写为 $P=R+(P-Q)+(Q-R)$。然后将$Q=R$代入，则得到$P=Q+(P-Q)+(Q-Q)$，由于$(Q-Q)=\vec{0}$，因此我们可以得到上述结果。■

vi. $(Q+\vec{v})-(R+\vec{w})=(Q-R)+(\vec{v}-\vec{w})$

【证明】$(Q+\vec{v})-(R+\vec{w})$

$=\left[(Q+\vec{v})-R\right]+\left[R-(R+\vec{w})\right]$	根据“首尾相接”公理
$=\left[(Q+\vec{v})-R\right]+\left[(R-R)-\vec{w}\right]$	根据（iv）式
$=\left[(Q+\vec{v})-R\right]-\vec{w}$	根据（i）式
$=\left[(Q+\vec{v})-Q\right]+[Q-R]-\vec{w}$	根据“首尾相接”公理
$=\left[\vec{v}+(Q-Q)\right]+[Q-R]-\vec{w}$	根据（iii）式
$=(Q-R)+(\vec{v}-\vec{w})$	根据（i）式 ■

仿射组合

我们已经讨论过如下的内容：

- 两个向量相加得到第三个向量
- 一个向量与一个标量相乘得到一个向量
- 一个向量与一个点相加得到另一个点
- 两个向量相减得到一个向量

注意我们还未讨论如下的内容：

- 一个点与标量相乘
- 将两个点相加

第一种运算在仿射空间中不存在合理的解析——缩放一个位置是什么意思？记住我们没有可区别的原点。第二种运算也不存在合理的解析。

然而，确实存在一种类似将两个点相加的运算，叫做仿射组合（affine combination）。

在一个仿射组合中，我们有效地将点的部分加在一起。在强烈地反对这种似乎比对点相加或缩放更古怪的运算之前，先考虑如下有两个点P和Q的情形。我们已经知道，这两个点之间的差可以用一个向量来表示，即$\vec{v} = Q - P$。因此，从P到Q之间存在无数的点，每一个点都对应$\vec{v}$的一个部分。考虑P和Q之间的一个任意点R。它将它们之间的向量分成一定的比例，假设为$\alpha : 1 - \alpha$。因此可写成

$$R = P + \alpha(Q - P)$$

图 3.25 用图形表示了这种关系。

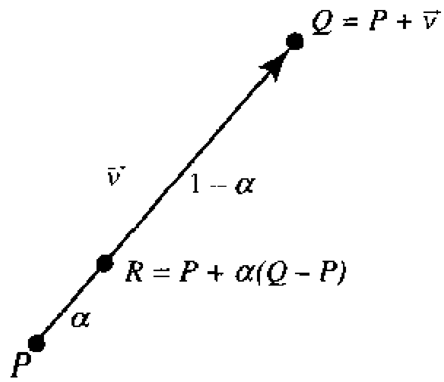


图 3.25　两个点的仿射组合

现在暂时假定我们是在进行代数运算，那么，上式可表示为

$$R = (1 - \alpha)P + \alpha Q$$

或者表示为

$$R = \alpha_1 P + \alpha_2 Q$$

其中$\alpha_1 + \alpha_2 = 1$。我们只是对点进行了两种“禁止的”运算——用一个向量来缩放点并将它们直接相加。可是，我们所做的只是“清除”了原来的仿射组合，因此在技术上并没有任何错误。然而，这种符号是如此方便，以至于已经成为通用的表示方法。抛开刚才的问题，我们可以做如下表述：只要如下表达式

$$\alpha_1 P + \alpha_2 Q$$

出现，并且$\alpha_1 + \alpha_2 = 1$，则它定义了如下的点

$$P + \alpha_2(Q - P)$$

当使用仿射组合这一术语时，上述形式被广泛地使用，同时它也是一种非常方便的表示法。

显然，如果将α设定在 0 和 1 之间，则点R将在P和Q之间。我们将这种仿射组合叫做凸组合。可是，我们关于仿射组合的定义并未真正地排除α超出该范围的情形，这将导致R位于P和Q所定义的（无限）线的某一位置。

与你所猜测的一样，我们可以将仿射组合扩展到多于两个点的情形：给定n个点$P_1, P_2, \ldots, P_n$，以及n个和为 1 的实数$\alpha_1, \alpha_2, \ldots, \alpha_n$，我们可以定义如下的仿射组合

$$P_1 + \alpha_2(P_2 - P_1) + \alpha_3(P_3 - P_2) + \cdots + \alpha_n(P_n - P_1)$$

并且也同样可以表示为

$$\alpha_1 P_1 + \alpha_2 P_2 + \cdots + \alpha_n P_n$$

图 3.26 的上图显示了一个实例，其中$\alpha_1 = \alpha_2 = \alpha_3 = \alpha_4 = 0.25$。仔细的读者可能已经注意到，$\alpha_1$没有出现在原来的仿射组合中，但它却出现在下面的重写形式中。为什么 P_1 和 α_1 特殊呢？其实它们并不特殊。可以将 P_1 与其他任何点交换，用同样的系数来计算仿射组合，我们将得到相同的点。图 3.26 的下图显示了交换 P_1 和 P_2的结果，我们得到了相同的仿射组合Q。

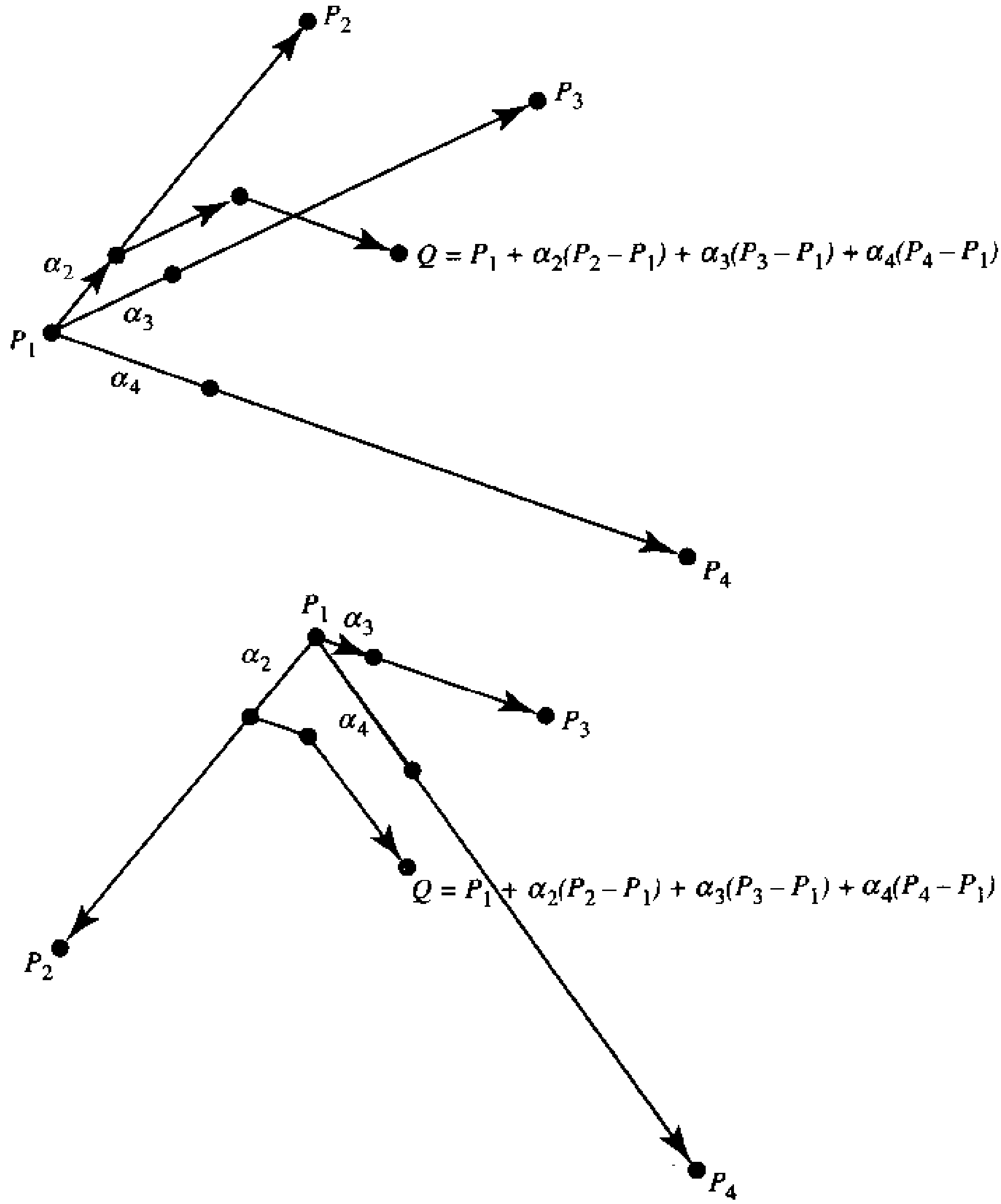


图 3.26　多点的仿射组合

3.3.1　欧几里得几何

你应该已经注意到，关于仿射几何的讨论缺少了一些内容：

- 在仿射空间中没有提及任何关于原点的概念。
- 我们只是泛泛地谈到角度，但并没有说明如何定义或计算角度。
- 仿射空间中的两个点显然被一定的距离所分开，但我们并未更多地讨论过距离。

这些内容并非无意地被遗漏了。实际上，根据定义，仿射空间中并不存在原点（不存在区别于其他点的特殊点），也不包括任何定义长度或角度的机制（记住，仿射空间本身由点所构成，因此，像“什么是两个点之间的角度”和“什么是点的长度”之类的问题，在仿射空间中是没有意义的）。

仿射空间中缺少预定义的原点并不会真正困扰我们，这是因为，在计算机图形学和几何设计中，模型（比如汽车等或虚拟世界）一般是通过组件层次来定义的，每一组件都在自己的空间中定义，它们的空间都位于比它更高一层的空间内，因此，没有任何点被真正地区分——只有点之间的相对关系是重要的。在稍后的一节中我们将讨论空间的原点问题。

现在我们集中讨论长度、距离和角度。这些概念并未因为它们不是有用的概念而从仿射空间中删除——它们当然是重要的。这些性质恰好是所谓的欧几里得空间的有效组成部分，欧几里得空间可被看成是加入了“尺寸”信息的仿射空间。因此，欧几里得空间也可被看做仿射空间的特殊化或子集；我们已经讨论过的仿射空间的所有原理和性质都适用于欧几里得空间，欧几里得空间还包含这些新的尺寸性质。

我们已知如何加减向量，如何用标量乘以向量，以及这两类运算是如何结合的。还有两种不同的基本向量运算，它们都是两个向量相乘的形式，即数量积（这样命名是因为它产生一个单一值结果，即数量）和向量积（这样命名是因为它产生另一个向量）。

数量积（scalar product）与如下的问题相关，即“什么是两个向量之间的角度”和“什么是向量的长度”（如图 3.27 所示），而向量积（vector product）与两个向量尾部与尾部相连（与加法运算的图形一样）所构成的平行四边形的面积相关（如图 3.28 所示）。



图 3.27　向量之间的角度和向量的长度

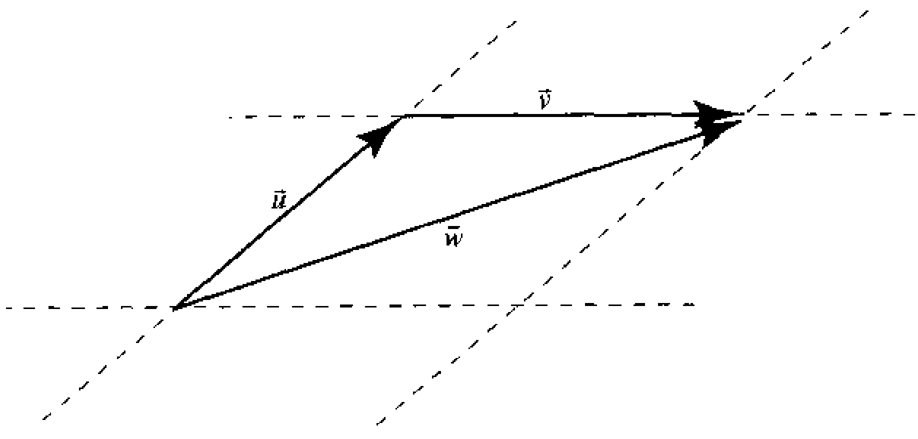


图 3.28　向量加法的平行四边形法则

1. 数量积

数量积通常又称为点积，是内积的一种特例。在继续讨论之前，我们需要先定义一些符号。

- 长度：向量$\vec{u}$的长度记为$\|\vec{u}\|$。
- 方向：向量$\vec{u}$的方向记为 dir（$\vec{u}$）。
- 符号：向量的符号记为 sgn（α）。
- 垂直：向量$\vec{u}$垂直于向量$\vec{v}$记为$\vec{u} \perp \vec{v}$。
- 平行：向量$\vec{u}$平行于向量$\vec{v}$记为$\vec{u} \parallel \vec{v}$。

在讨论数量积之前，我们需要退后一步，先讨论向量的投影，以提供关于数量积的用途和定义的一些直观知识。

假定有两个向量$\vec{u}$和$\vec{v}$，如图 3.29 一样将它们画出来，它们之间的角度是θ。向量$\vec{v}$ 相对于$\vec{u}$可以分解为两个分量：

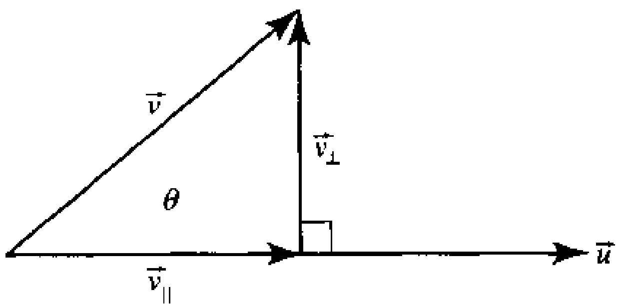


图 3.29　向量投影

- $\vec{v}_\perp$（垂直于$\vec{u}$）
- $\vec{v}_\parallel$（平行于$\vec{u}$）

注意，$\vec{v}_\parallel + \vec{v}_\perp = \vec{v}$。一般将$\vec{v}_\perp$叫做$\vec{v}$相对于$\vec{u}$的法分量，$\vec{v}_\parallel$ 叫做 $\vec{v}$ 在$\vec{u}$上的正交投影（称为"正交"是因为它是在垂直于$\vec{u}$的方向上的投影）。

我们感兴趣的是$\vec{v}_\parallel$，$\vec{v}_\perp$和角度θ之间的关系。我们首先应用一点三角知识来计算 $\vec{v}_\parallel$ 和$\vec{v}_\perp$的长度，并注意到（根据正弦和余弦的定义）

$$\|\vec{v}_\perp\| = \|\vec{v}\| |\sin\theta| \tag{3.2}$$

和

$$\|\vec{v}_\parallel\| = \|\vec{v}\| |\cos\theta| \tag{3.3}$$

那么，向量本身又如何呢？要回答这个问题需要更多的说明，仅仅运用基本的三角集合关系定义是不够的。第一个论断如下

$$\vec{v}_\parallel = \|\vec{v}\| \cos\theta \hat{u}$$

其中$\hat{u} = \frac{\vec{u}}{\|\vec{u}\|}$是与$\vec{u}$同向的单位向量（长度为 1）。即通过用$\vec{v}$和$\vec{u}$的长度之比来缩放$\vec{u}$，再乘以它们之间夹角的余弦就可以得到$\vec{v}_\parallel$。为了证明该论断确实如此，我们需要证明这两个向量是同一个向量，即证明它们具有相同的方向和长度。先看长度：

$$\|\vec{v}_\parallel\| = \|\vec{v} \cos\theta \hat{u}\|$$

$$\|\vec{v}_\parallel\| = \|\vec{v}\| |\cos\theta| \|\hat{u}\|$$

根据单位向量的定义，有$\|\hat{u}\|=1$，因此

$$\|\vec{v}_{\|}\| = \|\vec{v}\|\ |\cos\theta|$$

这就证明了这两个向量具有相同的长度。

为了说明它们是同向的，我们需要考虑如下两种情况：

i．$\cos\theta$为正（如图 3.29 所示）。

ii．$\cos\theta$为负（如图 3.30 所示）。

由于

$$\operatorname{dir}\left(\|\vec{v}\|\cos\theta\hat{u}\right) = \operatorname{dir}\left(\hat{u}\right) \Longleftrightarrow \cos\theta > 0$$

$$\operatorname{dir}\left(\|\vec{v}\|\cos\theta\hat{u}\right) = -\operatorname{dir}\left(\hat{u}\right) \Longleftrightarrow \cos\theta < 0$$

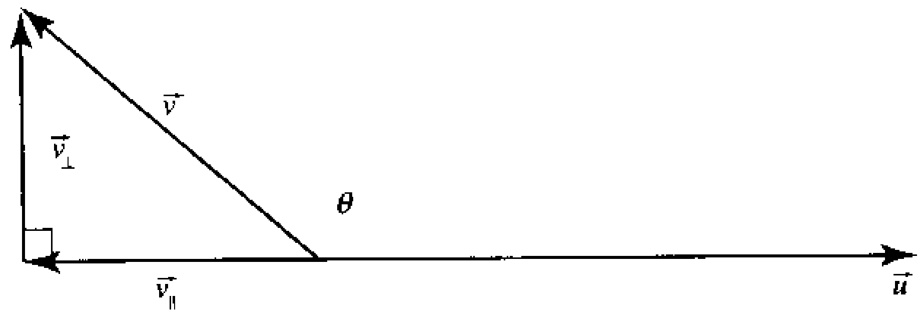


图 3.30　$\cos\theta$ 为负

所以，无论$\cos\theta < 0$还是$\cos\theta > 0$，我们都有

$$\operatorname{dir}\left(\|\vec{v}\|\cos\theta\hat{u}\right) = \operatorname{dir}\left(\vec{v}_{\|}\right) \tag{3.4}$$

注意，如果$\cos\theta = 0$（$\theta = 90°$或$\theta = 270°$），那么等式两边都是零向量，上述关系依然存在。

为了说明式（3.2），只需简单地注意到

$$\vec{v} = \vec{v}_{\perp} + \vec{v}_{\|}$$

它可以改写为

$$\vec{v}_{\perp} = \vec{v} - \vec{v}_{\|}$$

这样就可直接得出结论。

因此，在正交投影($\vec{v}_{\|}$)中有一个与两个向量之间的夹角相关的实体[1]。如果观察我们刚刚证明的关系和相关的图形，就可以注意到$\vec{v}$的长度和方向，以及$\vec{u}$的方向，都会影响$\vec{v}_{\|}$的长度和方向，但是$\vec{u}$的长度却对此没有影响！进一步可知，由于$\vec{v}_{\|}$是一个向量，因此在同时考虑两个向量的长度时，用一个数量——单一的值——来表示角度更合理。

上述论述说明了两个向量的点积（数量积）的一般意义。点积的正式定义如下：如果$\vec{v}$和$\vec{u}$是向量，它们之间的夹角为θ，那么点积$\vec{u}\cdot\vec{v}$ 定义为

1．我们能够随意地聚焦于$\vec{v}_{\|}$是因为我们已知$\vec{v}$，并且$\vec{v}_{\|}$和$\vec{v}_{\perp}$可以通过简单的向量减法求得。

$$\vec{u}\cdot\vec{v}=\begin{cases}\|\vec{u}\|\|\vec{v}\|\cos\theta, & \text{如果}\ \vec{u}\neq\vec{0}\ \text{且}\ \vec{v}\neq\vec{0}\\ 0, & \text{如果}\ \vec{u}=\vec{0}\ \text{或}\ \vec{v}=\vec{0}\end{cases} \tag{3.5}$$

对于非零向量$\vec{v}$和$\vec{u}$，当然有

$$\cos\theta=\frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\|\vec{v}\|} \tag{3.6}$$

和

$$\theta=\cos^{-1}\frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\|\vec{v}\|} \tag{3.7}$$

点积具有一些重要的性质。

i．定义：$\vec{u}\cdot\vec{v}=\|\vec{u}\|\|\vec{v}\|\cos\theta$.

ii．双线性：$\forall\alpha,\beta\in\mathbb{R}$, 且 $\forall\vec{u},\vec{v},\vec{w}\in\mathcal{A}.\mathcal{V}$

a．$(\alpha\vec{u}+\beta\vec{v})\cdot\vec{w}=\alpha(\vec{u}\cdot\vec{w})+\beta(\vec{v}\cdot\vec{w})$

b．$\vec{u}\cdot(\alpha\vec{v}+\beta\vec{w})=\alpha(\vec{u}\cdot\vec{v})+\beta(\vec{u}\cdot\vec{w})$

iii．正定性：

a．$\forall\vec{u}\in\mathcal{A}.\mathcal{V},\vec{u}\neq\vec{0},\vec{u}\cdot\vec{u}>0$

b．$\vec{0}\cdot\vec{0}=0$.

iv．交换性：$\vec{u}\cdot\vec{v}=\vec{v}\cdot\vec{u}$

【证明】

$$\begin{aligned}\vec{u}\cdot\vec{v}&=\|\vec{u}\|\|\vec{v}\|\cos\theta\\&=\|\vec{u}\|\|\vec{v}\|\cos(-\theta)\\&=\vec{v}\cdot\vec{u}\ \blacksquare\end{aligned}$$

v．点积对向量加法的分配性：$\vec{u}\cdot(\vec{v}+\vec{w})=(\vec{u}\cdot\vec{v})+(\vec{u}\cdot\vec{w})$

首先，我们必须证明一个简单的关系：$\vec{u}\cdot\vec{v}_{\|}=\vec{u}\cdot\vec{v}$。

$\vec{u}$与$\vec{v}_{\|}$之间的夹角α是0°或180°，取决于θ是否小于90°。因此，$\cos(\alpha)$相应地为1或−1，换一种说法可得$\cos(\alpha)=\mathrm{sgn}(\cos\theta)$。因此，我们有

$$\vec{u}\cdot\vec{v}_{\|}=|\vec{u}||\vec{v}_{\|}|\cos\alpha \tag{3.8}$$

$$=|\vec{u}||\vec{v}||\cos\theta|\ \mathrm{sgn}(\cos\theta) \tag{3.9}$$

$$=|\vec{u}||\vec{v}|\cos\theta \tag{3.10}$$

$$=\vec{u}\cdot\vec{v} \tag{3.11}$$

【证明】设γ为$\vec{v}_{\|}+\vec{w}_{\|}$和$\vec{u}$之间的夹角，通过式（3.8）可得

$$\begin{aligned}\vec{u}\cdot(\vec{v}+\vec{w})&=\vec{u}\cdot(\vec{v}+\vec{w})_{\|}\\&=\vec{u}\cdot(\vec{v}_{\|}+\vec{w}_{\|})\\&=\|\vec{u}\|\|\vec{v}_{\|}+\vec{w}_{\|}\|\cos\gamma\end{aligned}$$

有两种情形。

a．平行：$\vec{u}_{\|}\parallel\vec{v}_{\|}$

$$\begin{aligned}\vec{u}\cdot\left(\vec{v}+\vec{w}\right) &= \|\vec{u}\|\|\vec{v}_{\|}+\vec{w}_{\|}\|\cos\gamma \\ &= \|\vec{u}\|\left(\|\vec{v}_{\|}\|+\|\vec{w}_{\|}\|\right)\cos\gamma \\ &= \vec{u}\cdot\vec{v}_{\|}+\vec{u}\cdot\vec{w}_{\|} \\ &= \vec{u}\cdot\vec{v}+\vec{u}\cdot\vec{w}\end{aligned}$$

b．不平行：$\vec{u}_{\|}\nparallel\vec{v}_{\|}$

$$\begin{aligned}\vec{u}\cdot(\vec{v}+\vec{w}) &= \|\vec{u}\|\|\vec{v}_{\|}+\vec{w}_{\|}\|\cos\gamma \\ &= \|\vec{u}\|(\|\vec{v}_{\|}\|-\|\vec{w}_{\|}\|) \\ &= \|\vec{u}\|\|\vec{v}_{\|}\|-\|\vec{u}\|\|\vec{w}_{\|}\| \\ &= \vec{u}\cdot\vec{v}_{\|}+\vec{u}\cdot\vec{w}_{\|} \\ &= \vec{u}\cdot\vec{v}+\vec{u}\cdot\vec{w} \ \blacksquare\end{aligned}$$

vi．向量加法对点积的分配性：$(\vec{u}+\vec{v})\cdot\vec{w}=\vec{u}\cdot\vec{w}+\vec{v}\cdot\vec{w}$。

【证明】

$$\begin{aligned}(\vec{u}+\vec{v})\cdot\vec{w} &= \vec{w}\cdot(\vec{u}+\vec{v}) && \text{交换性} \\ &= \vec{w}\cdot\vec{u}+\vec{w}\cdot\vec{v} && \text{分配性} \\ &= \vec{u}\cdot\vec{w}+\vec{v}\cdot\vec{w} && \text{交换性} \ \blacksquare\end{aligned}$$

因此我们有

i．长度平方：$\vec{u}\cdot\vec{u}=\|\vec{u}\|^2$

ii．角度：$\theta=\cos^{-1}\frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\|\vec{v}\|}$

iii．投影：$\vec{v}_{\|}=\frac{(\vec{u}\cdot\vec{v})\vec{u}}{\vec{u}\cdot\vec{u}}$

iv．法线：$\vec{v}_{\perp}=\vec{u}-\frac{(\vec{u}\cdot\vec{v})\vec{u}}{\vec{u}\cdot\vec{u}}$

v．垂线：$\vec{u}\cdot\vec{v}=0\Longleftrightarrow\vec{u}\perp\vec{v}$

在本节讨论仿射组合时，我们解释了“滥用符号”，即允许用数量来乘以点。对于点乘的情形，我们有时也滥用符号，允许向量与点进行点积。例如，在 5.1.1 节中，我们将直线的隐含形式描述为$\vec{n}\cdot X=d$。自然，点积中涉及点并不是严格“合法的”，但是，正如仿射组合的情形，我们将抛开这些问题来做如下定义：如果出现$\vec{n}\cdot X$形式的表达式，其真正含义是$\vec{n}\cdot(X-\mathcal{O})$，其中$\mathcal{O}$是仿射面的原点。

2．向量积

向量的另一种乘法是向量积（vector product），又叫做叉积。我们在 n 维设置中讨论数积，但限定在三维中讨论叉积；将叉积扩展到更高的维时，仅有某些维数可行。与点积一样，叉积也与两个向量之间的角度相关，但也可理解为叉积定义了将两个向量首尾相接放置所确定的平行四边形的面积（如图 3.31 所示）。

研究叉积的另一种有用的方法是：如果有两个（不平行的）向量$\vec{u}$和$\vec{v}$，可以认为它们定义了一个（二维）平面。如果将该平面想像为“漂浮在空间中”，而不是“放在书页上”，那么，点积能帮助我们发现向量之间的角度，但并未提供任何关于平面在空间中的

方向的信息。

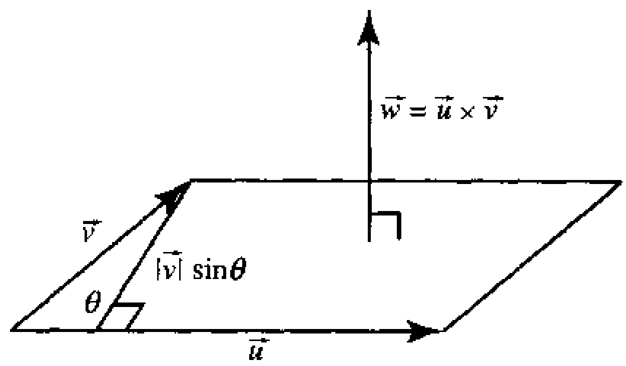


图 3.31　向量积

我们将两个向量$\vec{v}$和$\vec{v}$的叉积定义为另一个向量$\vec{w}$，它垂直于包含$\vec{u}$和$\vec{v}$的平面，其长度与这两个向量之间的交角相关。我们用符号“×”来表示叉积。其定义性质如下：

i．两个向量的叉积$\vec{w}=\vec{u}\times\vec{v}$是一个向量。

ii．两个向量的叉积垂直于这两个向量$\mathrm{dir}(\vec{u}\times\vec{v})\perp\vec{u},\vec{v}$。

iii．两个向量的叉积的长度等于它们所构成的平行四边形的面积：$\|\vec{u}\times\vec{v}\|=\mathrm{Area}\,(\vec{u},\vec{v})=\|\vec{u}\|\|\vec{v}\|\sin\theta$。

注意，如果$\theta>0$，则面积为正，如果$\theta<0$，则面积为负；如果想得到无符号的面积，则需要在公式中使用$\sin\theta$的绝对值。

精明的读者可能已经注意到，有两个垂直于$\vec{u}$和$\vec{v}$所决定的平面的向量，一个指向“外”或“上”，另一个与它相反，指向“内”或“下”。一般地，我们使用前面介绍过的右手法则：如果$\vec{u}$和$\vec{v}$的夹角是正的，则叉积指向“页外”，如果夹角为负，则反之（如图 3.32 所示）。

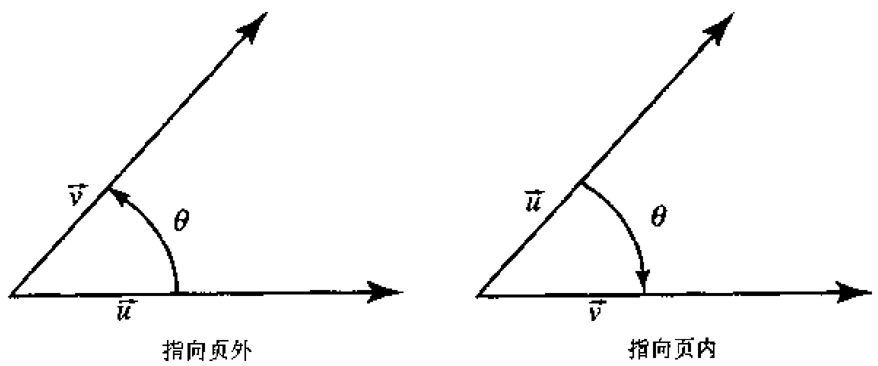


图 3.32　右手法则

以下是叉积的其他性质。

i．非交换性：$\vec{u}\times\vec{v}\neq\vec{v}\times\vec{u}$

ii．分配性：$\vec{u}\times(\vec{v}+\vec{w})=(\vec{u}\times\vec{v})+(\vec{u}\times\vec{w})$

iii．分配性：$(\alpha\vec{u})\times\vec{v}=\vec{u}\times(\alpha\vec{v})=\alpha(\vec{u}\times\vec{v})$

iv．平行律：$\vec{u}\parallel\vec{v}\Longleftrightarrow\vec{u}\times\vec{v}=\vec{0}$

3.3.2　体积、行列式和数量三重积

由于我们已经有了定义长度和面积的运算，因此可以推断，我们也能用向量运算来定义体积。自然，用于定义叉积的平行四边形的立体对应体就是平行六面体（如图 3.33 所示）。我们首先引入一个表示体积的符号：如果有一个由三个线性无关的向量，$\vec{u}, \vec{v}$ 和 $\vec{w}$ 定义的平行六面体，则其体积为

$$\text{Vol}\left(\vec{u}, \vec{v}, \vec{w}\right)$$

注意，它们的次序并不重要：

$$\text{Vol}\left(\vec{u}, \vec{v}, \vec{w}\right) = \text{Vol}\left(\vec{v}, \vec{w}, \vec{u}\right) = \cdots = \text{Vol}\left(\vec{w}, \vec{v}, \vec{u}\right)$$

因为它们都描述同一个平行六面体的体积。

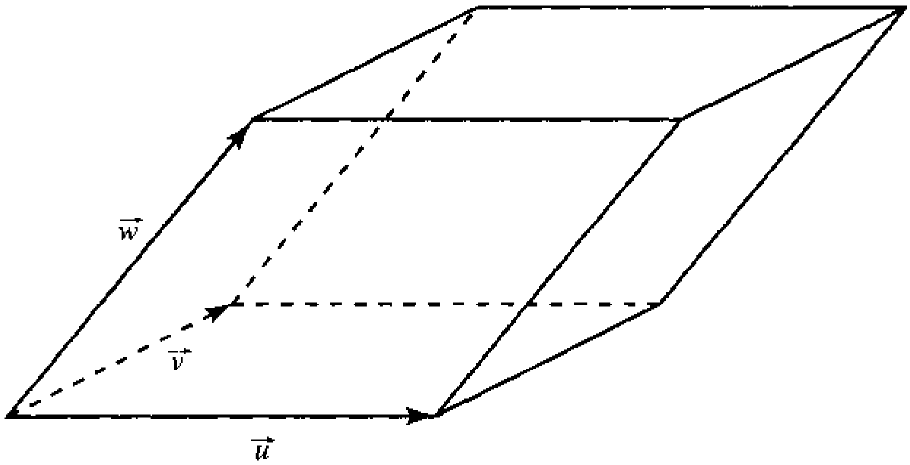


图 3.33　三个向量定义的平行六面体

那么，给定三个（基底）向量，如何确定 $\text{Vol}\left(\vec{u}, \vec{v}, \vec{w}\right)$？观察图 3.34。

$$\begin{aligned}
\text{Vol}\left(\vec{u}, \vec{v}, \vec{w}\right) &= \text{base} \times \text{height} && \text{定义} \\
&= \|\vec{u}\|\|\vec{v}\| \sin\psi \times \|\vec{w}\| \,|\cos\theta| && \text{三角几何} \\
&= \|\vec{u}\|\|\vec{v}\| \sin\psi \cdot \|\vec{w}_{\|}\| && \text{点积定义} \\
&= \|\vec{u} \times \vec{v}\| \cdot \|\vec{w}_{\|}\| && \text{叉积定义} \\
&= \|\vec{u} \times \vec{v} \cdot \vec{w}_{\|}\| && \text{长度定义} \\
&= \|\vec{u} \times \vec{v} \cdot \vec{w}\|
\end{aligned}$$

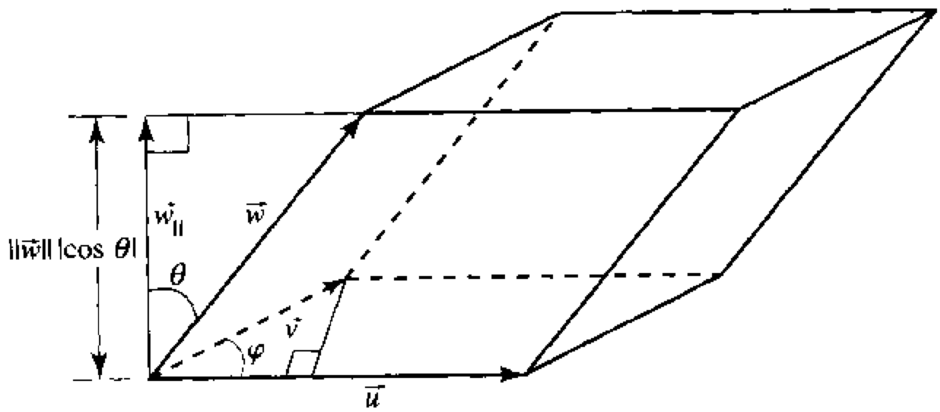


图 3.34　三重数积

但是，又如何确定方向呢？从上述说明中可以得出两个结论：

$$\mathrm{Vol}\left(\vec{u},\vec{v},\vec{w}\right)=\begin{cases}\left(\vec{u}\times\vec{v}\right)\cdot\vec{w} & \Longleftrightarrow \vec{w}_{\|}\parallel\vec{u}\times\vec{v}\\ -\left(\vec{u}\times\vec{v}\right)\cdot\vec{w} & \Longleftrightarrow \vec{w}_{\|}\parallel\vec{u}\times\vec{v}\end{cases}$$

根据右手法则，

$$\mathrm{sgn}\left(\vec{u},\vec{v},\vec{w}\right)=\mathrm{sgn}\left(\vec{u},\vec{v},\vec{w}_{\|}\right)=\begin{cases}+1 & \Longleftrightarrow \vec{w}_{\|}\parallel\vec{u}\times\vec{v}\\ -1 & \Longleftrightarrow \vec{w}_{\|}\parallel-\vec{u}\times\vec{v}\end{cases}$$

因此可以得出如下结论

$$\mathrm{Vol}\left(\vec{u},\vec{v},\vec{w}\right)=\mathrm{sgn}\left(\vec{u},\vec{v},\vec{w}\right)\left(\left(\vec{u}\times\vec{v}\right)\cdot\vec{w}\right) \tag{3.12}$$

表达式

$$\left(\vec{u}\times\vec{v}\right)\cdot\vec{w} \tag{3.13}$$

通常用来表示三重数量积，就是行序（或列序）为 $\vec{u},\vec{v},\vec{w}$ 的矩阵的行列式，表示为

$$\det\left(\vec{u},\vec{v},\vec{w}\right)$$

在上面的讨论中，我们将符号分离开来，以强调行列式是有符号的体积。

注意，上述行列式的定义式（3.13）是式（3.12）的子式，即行列式是有符号的体积。

$$\mathrm{Vol}\left(\vec{u},\vec{v},\vec{w}\right)=\left|\det\left(\vec{u},\vec{v},\vec{w}\right)\right|=\mathrm{sgn}(\vec{u},\vec{v},\vec{w})\det\left(\vec{u},\vec{v},\vec{w}\right)$$

与行列式、三重数量积和体积相关的其他性质如下（Goldman 1987）。

i．只有当向量集$\{\vec{u},\vec{v},\vec{w}\}$构成一个基底时，行列式$\det\left(\vec{u},\vec{v},\vec{w}\right)$才是非零的。例如，在三维空间中，如果三个向量不构成一个基底，那么它们只能生成一个面或者一条线，而这两者都没有体积。

ii．只有当$\{\vec{u},\vec{v},\vec{w}\}$的符号为正时，行列式$\det\left(\vec{u},\vec{v},\vec{w}\right)$才是正的。

iii．循环改变向量的次序并不会改变其行列式：

$$\det(\vec{u},\vec{v},\vec{w})=\det(\vec{w},\vec{u},\vec{v})=\det(\vec{v},\vec{w},\vec{u})$$

iv．逆转向量的次序将改变行列式的符号，但不会改变其数量：

$$\det(\vec{u},\vec{v},\vec{w})=-\det(\vec{w},\vec{v},\vec{u})=-\det(\vec{v},\vec{u},\vec{w})=-\det(\vec{u},\vec{w},\vec{v})$$

v．反转任何一个向量都将改变行列式的符号：

$$\det(\vec{u},\vec{v},\vec{w})=-\det(-\vec{u},\vec{v},\vec{w})=-\det(\vec{u},-\vec{v},\vec{w})=-\det(\vec{u},\vec{v},-\vec{w})$$

vi．缩放向量将导致缩放行列式：

$$\det(c\vec{u},\vec{v},\vec{w})=\det(\vec{u},c\vec{v},\vec{w})=\det(\vec{u},\vec{v},c\vec{w})=c\det(\vec{u},\vec{v},\vec{w})$$

vii．右手标准正交空间的基底向量的行列式是单位行列式。

3.3.3 坐标系

我们现在已经具备了讨论仿射空间坐标的条件。前面已经讲过，仿射空间 $\mathcal{A}$ 定义为点集$\mathcal{P}$（一个点空间）加上相关或潜在的向量空间$\mathcal{V}$，它们都具有相同的维数n。如果我们选取任意一点$\mathcal{O}\in\mathcal{P}$和一个基底$\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n\in\mathcal{V}$，那么这就构成了$\mathcal{A}$ 的一个坐标系。可

以将坐标系写成

$$\mathcal{F} = (\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n, \mathcal{O})^{\mathrm{T}}$$

回想一下，在向量空间中，任何向量都可写成一组基底向量的一个线性组合（3.2.3 节）。任何 $\vec{u} \in \mathcal{V}$ 都可写成

$$\vec{u} = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n$$

其中 $a_1, a_2, \ldots, a_n$ 是相对于基底 $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ 的坐标。

那么，点集 $\mathcal{P}$ 又具有什么意义呢？向量和点空间在此汇集在一起。回忆一下，对任何点 P 和任何向量 $\vec{u}$，都存在一个惟一的点 $Q = P + \vec{u}$。如果我们从 $\mathcal{F}$ 中选择 $\mathcal{O}$ 作为 P，那么任意点 $Q \in \mathcal{P}$ 都可以被定义为一个惟一向量 $\vec{u} = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n$ 与 $\mathcal{O}$ 之和：

$$\begin{aligned} Q &= \vec{u} + \mathcal{O} \\ &= a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n + \mathcal{O} \end{aligned}$$

同样，Q 的坐标为 $a_1, a_2, \ldots, a_n$。图 3.35 显示了仿射空间 $\mathcal{A} = (\mathcal{P}, \mathcal{V})$ 及其坐标系 $\mathcal{F} = (\vec{v}_1, \vec{v}_2, \mathcal{O})^{\mathrm{T}}$；点 Q 为 $\mathcal{O} + \vec{u}$，其坐标为（3,2）。

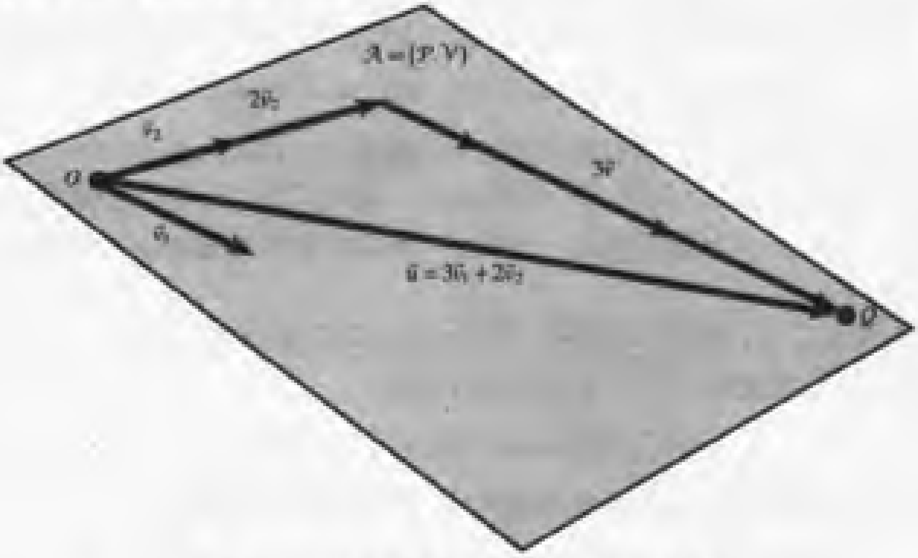


图 3.35　一个仿射点相对于任意坐标系的坐标

笛卡尔坐标系

注意前面我们已经提及坐标，并且选取 $\mathcal{O}$ 时至少暗示了一个相对的原点。然而，到目前为止，我们对坐标系和其潜在的基底向量的要求仅仅是它们是线性无关的。只有在引入了点积之后，我们才有了正式定义和描述角度和长度的方法。我们现在就利用角度和长度的点积的定义性质来定义一个欧几里得空间的特殊子集。

每一个向量 $\vec{v}$ 都有一个与其相关的单位向量，记为 $\hat{v}$。它的方向与 $\vec{v}$ 相同，但长度为 1：

$$\hat{v} = \frac{\vec{v}}{\|\vec{v}\|}$$

即仅仅用其自身的长度来缩放 $\vec{v}$。

由于我们能测量和定义角度，因此我们能通过要求 $\vec{v}_1 \cdot \vec{v}_2 = 0$ 来保证基底向量是垂直的

或正交的。如果有一个三维空间，而且三个基底向量是相互垂直的，那么，其中任何一个向量都是其余两个向量的叉积（按右手法则来确定次序 $\vec{v}_1 \times \vec{v}_2 = \vec{v}_3$，$\vec{v}_2 \times \vec{v}_3 = \vec{v}_1$，$\vec{v}_3 \times \vec{v}_1 = \vec{v}_2$）。

有了这些工具，我们可以在欧几里得空间中定义一种特殊类型的坐标系——笛卡尔坐标系，其基底向量具有单位长度，并且相互垂直。这样的基底也称为标准正交。

3.4 仿射变换

仿射变换（affine transformation）是将一个仿射空间中的点和向量分别对应到另一个仿射空间中的点和向量的映射。一般地，我们将 $T : \mathcal{A}^n \mapsto \mathcal{B}^m$ 叫做仿射变换，如果它保持如下的仿射组合：

$$T\left(a_1 P_1 + a_2 P_2 + \cdots + a_n P_n\right) = a_1 T\left(P_1\right) + a_2 T\left(P_2\right) + \cdots + a_n T\left(P_n\right) \tag{3.14}$$

其中 $P_i \in \mathcal{A}$ 且 $\sum_{i=1}^{n} a_i = 1$。注意维数 n 和 m 并不要求相等，但要求 $m \leqslant n$。

由于仿射变换将点映射到点，因此它也将线段映射到线段，将面映射到面，等等。我们可以更直接地显示这一点：已知可以将一条线上的点 R 写成该线上的（不同的）两个其他点 P 和 Q 的仿射组合

$$R = (1-\alpha)\,P + \alpha Q$$

如果应用仿射映射 T，可得

$$\begin{aligned} T(R) &= T((1-\alpha)\,P + \alpha Q) \\ &= (1-\alpha)\,T(P) + \alpha T(Q) \end{aligned}$$

该式与参数形式的直线方程关系密切：

$$R(t) = (1-t)\,P + tQ$$

对其应用映射 T：

$$\begin{aligned} T(R(t)) &= T((1-t)\,P + tQ) \\ &= (1-t)\,T(P) + tT(Q) \end{aligned}$$

这就是由点 P 和 Q 所定义的直线的参数方程。虽然这是显而易见的，但是指出来还是非常重要的，即上面方程中的常数并不受变换 T—t 的影响，而且 $1-t$ 不会改变。换句话说，R 与 P 和 Q 的相对距离相等。正规地说，仿射映射保持相对比例（如图 3.36 所示）。

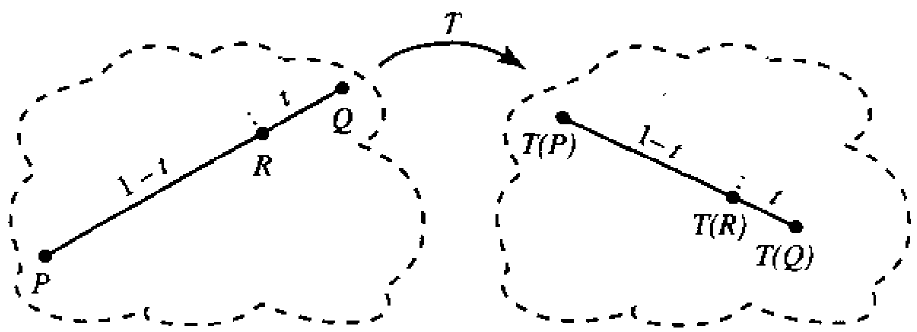


图 3.36 仿射映射保持相对比例

正如你所知道的一样，仿射空间是一个点集加上一个向量集。因此，这很自然地引起一个疑问，即对仿射映射的影响将如何影响向量。假定我们有一个对仿射空间$\mathcal{A}$中的点的仿射映射T。给定两个点$P, Q \in \mathcal{A}$，我们可用它们的差来定义一个向量

$$\vec{v} = Q - P$$

这是因为这种操作定义了仿射空间中点与向量的关系。那么，如果我们对其应用仿射变化，结果会怎样呢？我们将得到

$$\begin{aligned} T(\vec{v}) &= T(Q - P) \\ &= T(Q) - T(P) \end{aligned}$$

因此，变换所得的向量就是变换所得的两个点所确定的变量。我们还知道，仿射空间中的向量也是向量空间中的元素，而且在向量空间中，向量的位置是没有意义的。$T(\mathcal{A})$中存在无数个差为$T(\vec{v})$的点对；如果在每一个这样的点对之间画一条有向线段，那么就"复制"了一些具有相同的方向和数量的$T(\vec{v})$，它们之间具有一定的偏移量，或者从一个到另一个之间存在一些平移。

由此更进一步，我们可知仿射映射保持平行性。为了说明这点，假定有两个点对$\{P_1, P_2\}$和$\{Q_1, Q_2\}$，每个点对都定义了一条直线：

$$\mathcal{L}_1 = P_1 + \alpha(P_2 - P_1)$$

$$\mathcal{L}_2 = Q_1 + \beta(Q_2 - Q_1)$$

如果$P_2 - P_1 = \gamma(Q_2 - Q_1)$，则这两条直线是平行的（即这两个向量的方向相同，而它们的长度相差一个比率γ）。仿射映射将这些向量映射为相同向量的缩放版本，所以仿射映射保持平行性。

我们将该结果用来描述仿射变换就可得出如下结论：仿射映射T是关于仿射空间$\mathcal{A}$的线性变换。在 3.2.6 节中，我们将线性变换定义为保持线性组合的变换。向量的线性组合定义为：

$$\vec{w} = x_1\vec{v}_1 + x_2\vec{v}_2 + \cdots + x_n\vec{v}_n, x_i \in \mathbb{R}$$

其中$\vec{v}_i \in \mathcal{V}$为一组线性无关的向量。一个保持线性组合的线性映射必须满足

$$T(\vec{w}) = T\left(\vec{v}_1x_1 + \vec{v}_2x_2 + \cdots + \vec{v}_nx_n\right) \tag{3.15}$$

$$= T\left(\vec{v}_1x_1\right) + T\left(\vec{v}_2x_2\right) + \cdots + T\left(\vec{v}_nx_n\right) \tag{3.16}$$

$$= x_1T\left(\vec{v}_1\right) + x_2T\left(\vec{v}_2\right) + \cdots + x_nT\left(\vec{v}_n\right) \tag{3.17}$$

对$\forall x_1, x_2, \ldots, x_n \in \mathbb{R}, \forall \vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n \in \mathcal{V}$成立。为了分别说明保持线性组合的两个方面，以及相应地说明需要满足的两个条件，我们将上面的方程（3.15）分解为方程（3.16）和（3.17）。我们可以在二维仿射空间这样做，也可以通过归纳基底向量的数目将其扩展到更高维的空间。首先，我们必须说明

$$T(\vec{u} + \vec{v}) = T(\vec{u}) + T(\vec{v}) \tag{3.18}$$

假定有两个向量$\vec{u}$和$\vec{v}$，如图 3.37 所示。证明非常简单：

$$
\begin{aligned}
T(\vec{u}+\vec{v}) &= T(R-P) \\
&= T(R)+(T(Q)-T(Q))-T(P) \\
&= (T(R)-T(Q))+(T(Q)-T(P)) \\
&= T(\vec{v})+T(\vec{u}) \\
&= T(\vec{u})+T(\vec{v})
\end{aligned}
$$

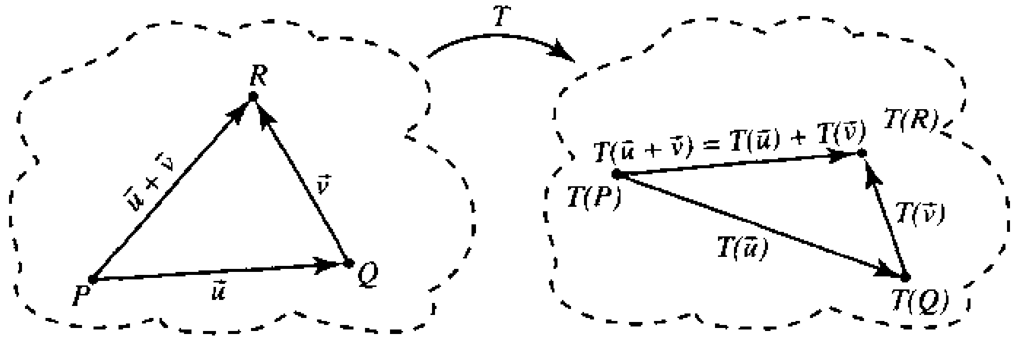


图 3.37　向量和

现在我们说明

$$T(\alpha\vec{v})=\alpha T(\vec{v})$$

如图 3.38 所示，我们可将 $\alpha\vec{v}$ 改写为$((1-\alpha)P+\alpha Q)-P$。如果“用图形证明”还不够充分，则考虑如下的证明：

$$
\begin{aligned}
T(\alpha\vec{v}) &= T(((1-\alpha)P+\alpha Q)-P) && \text{替换} \\
&= T((1-\alpha)P+\alpha Q)-T(P) && \text{方程 (3.18)} \\
&= (1-\alpha)T(P)+\alpha T(Q)-T(P) && \text{方程 (3.18)} \\
&= \alpha T(\vec{v}) && \text{仿射组合的定义}
\end{aligned}
$$

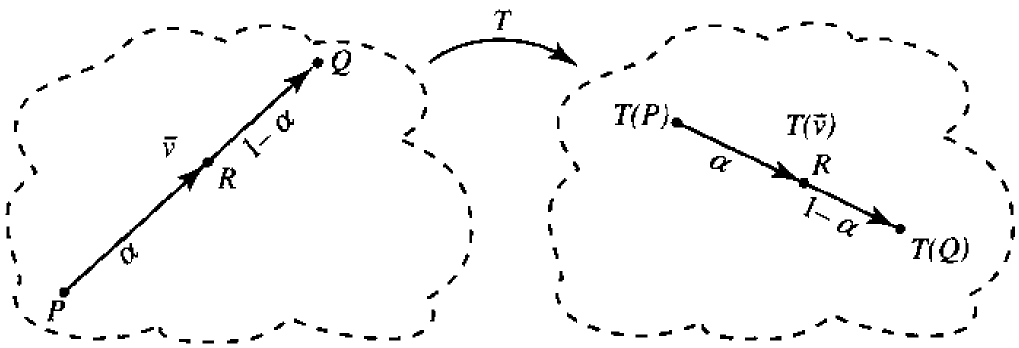


图 3.38　向量缩放

最后，仿射变换还保持向量和点的加法：

$$T(P+\vec{v})=T(P)+T(\vec{v})$$

从图 3.39 中可以看出一个点与向量的和确定一个点的一般定义：$Q=P+\vec{v}$ 或 $\vec{v}=Q-P$。据此可导出如下的证明（DeRose 1992）：

$$
\begin{aligned}
T(P+\vec{v}) &= T(P+(Q-P)) && \text{点的减法定义} \\
&= T(P)+T(Q)-T(P) && \text{变换的定义}
\end{aligned}
$$

$$= \quad T(P) + (T(Q) - T(P)) \quad \text{向量加法的结合性}$$
$$= \quad T(P) + T(\vec{v}) \quad \text{点的减法定义}$$

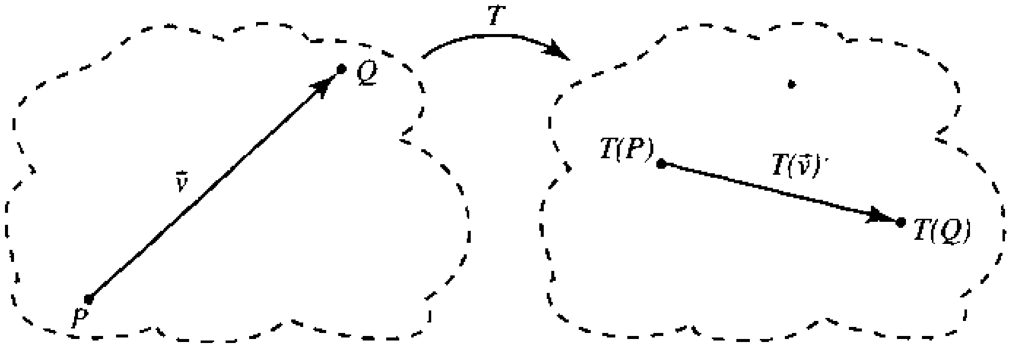


图 3.39　点与向量的和

综合以上性质可知，仿射变换 T 保持仿射坐标：

$$T\left(\alpha_1\vec{v}_1 + \alpha_2\vec{v}_2 + \cdots + \alpha_n\vec{v}_n + \mathcal{O}\right) = \alpha_1 T(\vec{v}_1) + \alpha_2 T(\vec{v}_2) + \cdots + \alpha_n T(\vec{v}_n) + T(\mathcal{O})$$

上式是一个通用的表达式，但应该注意，该式是针对仿射坐标系的。因此，仿射变换完整且惟一地由其对一个坐标系或单形的行为所定义。

3.4.1　仿射映射的类型

正如上节所说明的，仿射映射对仿射空间 $\mathcal{A}$ 中的向量的操作是一种线性映射；它允许旋转和缩放（均衡和非均衡）。由于向量（即使是 $\mathcal{A}.\mathcal{V}$ 中的向量）不包含位置信息，所以它排除了任何与位置相关的操作（比如平移）。

由于仿射变换同时针对 $\mathcal{A}.\mathcal{P}$ 和 $\mathcal{A}.\mathcal{V}$，将点映射到点，等等，因此这种变换能表示与相对位置有关的变换：

- 平移
- 相对任意线或面的镜像或反射
- 平行投影
- 相对任意点的旋转
- 相对任意线或面的剪切

4.7 节将更详细地介绍这些变换。

3.4.2　仿射映射的合成

在 2.7.1 节中，我们讨论了映射的一般概念，以及如何通过将一个函数 T 的输出（值域）作为另一个函数 U 的定义域来合成映射。仿射映射当然也具有相同的性质；我们可以通过将一个仿射映射用于另一个仿射映射的方式来建立一个复杂的变换系。前几节描述的仿射映射的性质保证了仿射映射不会离开仿射空间，因此，可以将任意数量的仿射映射的组合看成另一个单一的仿射映射，尽管那是一个更复杂的仿射映射。一个相当明显的例子是，一系列旋转角度为 α, β, γ（相对于同一点）的旋转，显然这相当于一个单一的旋转 $\alpha + \beta + \gamma$（如图 3.40 所示）。

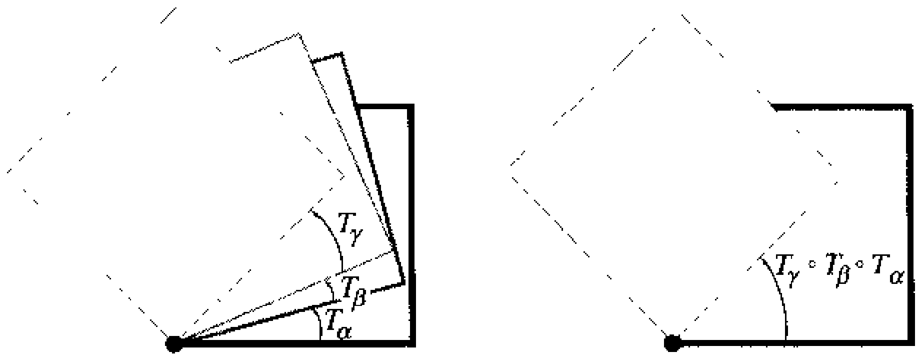


图 3.40　仿射映射的合成（旋转）

3.5　重心坐标和单形

我们知道，仿射空间中点的坐标可用向量空间的基底向量相对于$\mathcal{F}$的点 $\mathcal{O}$ 来定义：

$$\begin{aligned} Q &= \vec{u} + \mathcal{O} \\ &= a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n + \mathcal{O} \end{aligned}$$

另一种方法是使用所谓的"基底点"来定义：$P_0 = \mathcal{O}, P_1 = \mathcal{O} + \vec{v}_1, P_2 = \mathcal{O} + \vec{v}_2, \ldots, P_n = \mathcal{O} + \vec{v}_n$（即一个包含 $\mathcal{O}$ 和通过将基底向量与 $\mathcal{O}$ 相加而产生的点的点集）。

我们可以将点$Q \in \mathcal{A}$相对于$\mathcal{F}$表示为

$$Q = P_0(1 - a_1 - a_2 - \cdots - a_n) + P_1a_1 + P_2a_2 + \cdots + P_na_n$$

或

$$Q = P_0a_0 + P_1a_1 + \cdots + P_na_n$$

其中 a_0 由下式定义

$$1 = a_0 + a_1 + \cdots + a_n$$

最后一个恒等式尤其重要——系数之和为 1。

这可以确认为一个仿射组合，值$a_0, a_1, \ldots, a_n$ 叫做 Q 关于$\mathcal{F}$的重心坐标。图 3.41 显示了标准的坐标系坐标和重心坐标。

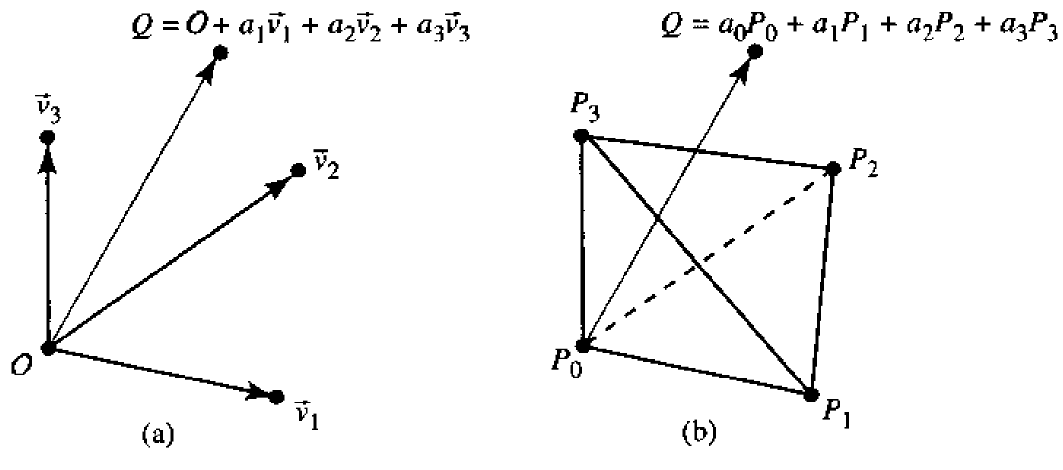


图 3.41　仿射坐标（a）和重心坐标（b）

由于仿射空间中点和向量的基本关系，你可能认为向量本身也能用重心坐标来表示。事实确实如此。我们知道，任何向量都可写成

$$\vec{u}=a_1\vec{v}_1+a_2\vec{v}_2+\cdots+a_n\vec{v}_n$$

如果设$a_0=-(a_1+a_2+\cdots+a_n)$，则可以将向量改写为

$$\vec{u}=a_0P_0+a_1P_1+\cdots+a_nP_n$$

注意

$$\begin{aligned}a_0+a_1+\cdots+a_n&=-(a_1+a_2+\cdots+a_n)+a_1+a_2+\cdots+a_n\\&=0\end{aligned}$$

即系数之和为 0，而不像点的情形一样为 1。“基底点”一般叫做单形（simplex），这与特定的点加上基底向量叫做坐标系类似。

仿射映射保持应用于更高阶单形的重心坐标，以及直线的相对比率（参见 3.4 节）。我们对此做进一步的推导。一个单形的基底点K就是重心坐标为$(a_0=0,a_1=0,\ldots,a_k=1,\ldots,a_n=0)$的点。因此，如果基底点被变换，我们将得到另一组定义另一个单形的基底点，其仿射组合等价于已应用仿射映射的点。所以，仿射映射可以被其对单形的操作所完全且惟一地描述。然而，这也说明，仿射映射甚至比这里所说的更通用。它可以将一个n维单形变换成一组非单形的n维点。当映射为投影时就是如此。

3.5.1 重心坐标和子空间

正如我们可以有线性（向量）子空间一样，我们也可以有仿射子空间，并且可以通过它们来讨论重心坐标。假定我们有一个用单形$S=(P_0,P_1,\ldots,P_n)$所定义的n维仿射空间$\mathcal{A}$。那么，我们就能定义一个由单形$\mathcal{T}=(Q_0,Q_1,\ldots,Q_m)$所定义的$m$维的子空间$\mathcal{B}\subset\mathcal{A}$。任意点$R\in\mathcal{B}$都可以表示为

$$R=b_0Q_0+b_1Q_1+\cdots+b_mQ_m$$

并具有定义$1=b_0+b_1+\cdots+b_m$。当然，由于Q_i是用$\mathcal{A}$来表示的，我们也可以用$\mathcal{B}$来改写R。

由于每一个单形都由$n+1$点所构成的，所以 1 维单形是一条线段，2 维单形是一个三角形（定义一个面），3 维单形是一个四面体（定义一个立体），如图 3.42 所示。该图也显示了重心坐标与标架坐标系之间的关系。考虑图中间的 2 维单形：如上所述，点R可用重心坐标来定义：从Q_0出发的一条线段与单形的对边相交于点c（Q_2-Q_1）（其他两个基底点也类似），并且，我们可将任何的Q_i看成定义仿射坐标系的$\mathcal{O}$和从该点到其邻点的向量。注意到任何两个内点，相交的线段都足以定义R。这也说明，只需两个单形系数就足以定义一个点：这一点之所以成立是因为这些系数之和为 1，所以，如果我们已知两个系数，第三个系数也就确定了。

3.5.2 仿射无关

对一个仿射坐标系来说，其基底向量必须是线性无关的。考虑到仿射坐标系或单形可

用来定义仿射空间，假定单形也存在类似的规范是合乎逻辑的。

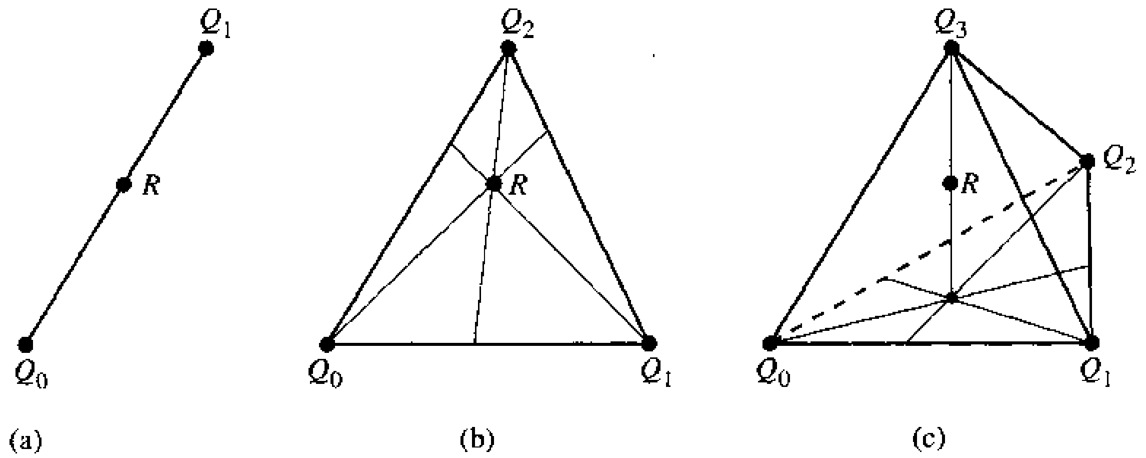


图 3.42　最基本的三种单形：直线（a），三角形（b）和四面体（c）

我们知道，向量的线性无关意味着它们是互不平行的。显然，对基底点来说，类似的性质就是它们是互不等同的，并且不存在多于两个的点是共线的。即任何一点都不是其他点的仿射组合。正规地说，如果一组基底点的单形坐标是线性无关的（同样，向量空间中的向量是线性无关的），则它们是仿射无关的。

假定 $P_0, P_1, \ldots, P_n$ 是定义 n 维单形的 $n+1$ 个点，并且 $\vec{v}_i = P_i - P_0$（记住我们采用如下惯例：$P_0 = \mathcal{O}$）。如果 n 个向量 $\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n$ 是线性无关的，那么点 $P_0, P_1, \ldots, P_n$ 是仿射无关的。从图 3.42 中可以看出这一点：定义一维单形的两个点不能是同一点；定义二维单形的三个点不能全部共线；定义三维单形的四个点不能共面。注意，如果我们像这样“退化”任何单形，我们可以得到维数少于 1 的空间，对应于“最接近的次阶”单形。

第 4 章　矩阵、向量代数和变换

4.1　导言

上一章的要点是使用不需要坐标的方法来介绍几何学的概念和原理。例如，大部分对点积的描述都只是简单地说明了如何对行矩阵和列矩阵执行算术运算来实现点积，并不提供更多的直觉的理解和解释，而我们采用的却是（直观的）纯几何学方法。

DeRose（1989，1992）和 Goldman（1985，1987）极力提倡这种不需要坐标的方法。DeRose 描述了一套不需要坐标的应用程序接口，并且已有了一套实现代码。这种方法是很值得推荐的，特别是与通常使用的方法相比较而言，常用的方法要求程序员明确地运算矩阵乘法、求逆，总要弄清楚“在什么空间中”，并利用坐标来进行所有的运算。

另一方面，实际情形是大部分的图形学软件并不是这样编写的，而且程序员需要处理矩阵并对它们执行运算；甚至即使是不需要坐标的工具库，也可能在其实现中使用矩阵。

本章的目的是将上一章介绍的概念和技术与其前一章介绍的矩阵结合在一起。

我们在第 2 章中将矩阵作为一种非常抽象的工具，使其脱离了与第 3 章介绍的向量代数的关系。然而，我们已经在其中留下了一些暗示，比如，我们把$1 \times n$矩阵叫做“行向量”，在介绍仿射变换时讨论坐标；学过变换、空间和矩阵的读者可能已经很清楚地了解了这一点，新接触这部分内容的读者也可能正在建立这种联系。

现在我们将所有这些概念结合在一起，并用“向量代数”的观点来明确地说明如何用矩阵来表示点、向量和变换。这与通常的处理方式不同，例如，在 Rogers 和 Adams（1990）或 Newman 或 Sproull（1979）的著作中，一般都是从用x，y和z坐标来描述点和向量开始介绍，将点积描述为不直观和相当任意的坐标运算，而变换被描述为魔术般的矩阵与点或向量的乘法。

4.2　点和向量的矩阵表示

在 3.3.3 节中，我们说明了仿射坐标系$\mathcal{F}$可用一组基底向量和一个原点来表示：

$$\mathcal{F} = (\vec{v}_1, \vec{v}_2, \ldots, \vec{v}_n, \mathcal{O})^{\mathrm{T}}$$

其中任何的$\vec{u} \in \mathcal{V}$（$\mathcal{V}$为向量空间）可表示为

$$\vec{u} = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n \tag{4.1}$$

并且其中的任何的点$P \in \mathcal{P}$（与坐标系的相关向量空间相联系的点的集合）都可表示为

$$P = a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n + \mathcal{O} \tag{4.2}$$

我们知道，3.3 节中定义的坐标公理指出，一个点与0相乘产生零向量，可表示为 $0 \cdot \mathcal{O} = \vec{0}$。将 2.3.4 节中定义的多元组乘法与其矩阵表示联系起来，我们可以用矩阵形式来改写方程（4.1）：

$$\begin{aligned}
\vec{u} &= a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n \\
&= a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n + (0 \cdot \mathcal{O}) \\
&= [a_1 \quad a_2 \quad \ldots \quad a_n \quad 0][\vec{v}_1 \quad \vec{v}_2 \quad \ldots \quad \vec{v}_n \quad \mathcal{O}]^{\mathrm{T}} \\
&= [a_1 \quad a_2 \quad \cdots \quad a_n \quad 0]\begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vdots \\ \vec{v}_n \\ \mathcal{O} \end{bmatrix} \\
&= [a_1 \quad a_2 \quad \cdots \quad a_n \quad 0]\begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,n} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n,1} & v_{n,2} & \cdots & v_{n,n} \\ \mathcal{O}_1 & \mathcal{O}_2 & \cdots & \mathcal{O}_n \end{bmatrix}
\end{aligned} \tag{4.3}$$

因此，我们可以将向量表示为前面n个元素为仿射坐标的系数、最后一个元素为0的行矩阵。如果上下文中清晰地说明了仿射坐标系，则可以使用缩写形式 $\vec{u} = [a_1 \quad a_2 \quad \cdots \quad a_n \quad 0]$。注意这种写法只是用来方便地表示向量$\vec{u}$和它在坐标系中的表示，但是等式的意义如方程 4.3 所示。

我们可以对点做相同的推论。再次引用坐标公理，我们将方程（4.2）改写成矩阵形式：

$$\begin{aligned}
P &= a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n + \mathcal{O} \\
&= a_1\vec{v}_1 + a_2\vec{v}_2 + \cdots + a_n\vec{v}_n + (1 \cdot \mathcal{O}) \\
&= [a_1 \quad a_2 \quad \ldots \quad a_n \quad 1][\vec{v}_1 \quad \vec{v}_2 \quad \ldots \quad \vec{v}_n \quad \mathcal{O}]^{\mathrm{T}} \\
&= [a_1 \quad a_2 \quad \cdots \quad a_n \quad 1]\begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vdots \\ \vec{v}_n \\ \mathcal{O} \end{bmatrix} \\
&= [a_1 \quad a_2 \quad \cdots \quad a_n \quad 1]\begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,n} \\ v_{2,1} & v_{2,2} & \cdots & v_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n,1} & v_{n,2} & \cdots & v_{n,n} \\ \mathcal{O}_1 & \mathcal{O}_2 & \cdots & \mathcal{O}_n \end{bmatrix}
\end{aligned} \tag{4.4}$$

因此，我们可以将点表示为前面的n个元素为仿射坐标的系数、最后一个元素为 1 的行矩阵。如果上下文中清晰地说明了仿射坐标系，则可以使用缩写形式$P = [a_1 \quad a_2 \quad \cdots \quad a_n \quad 1]$。注意这种写法只是用来方便地表示点$P$和它在坐标系中的表示，但是等式的意义如方程

（4.4）所示。

当然，仿射坐标系$\mathcal{F}$的基底向量和原点与仿射空间中的其他向量和点没有什么不同，因此可以用矩阵来表示它们

$$\begin{bmatrix} v_{1,1} & v_{1,2} & \cdots & v_{1,n} & 0 \\ v_{2,1} & v_{2,2} & \cdots & v_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ v_{n,1} & v_{n,2} & \cdots & v_{n,n} & 0 \\ \mathcal{O}_1 & \mathcal{O}_2 & \cdots & \mathcal{O}_n & 1 \end{bmatrix} \tag{4.5}$$

可以看出，这是一个$(n+1)\times(n+1)$的方阵。在如下的讨论中可以看出，这种形式是很方便的。

具有二维或三维图形编程经验的读者可能已经很熟悉，我们一般仅用坐标来表示点或向量。对此做一些解释是合适的：通常用某种分层的方式来表示单独的对象或整个场景。就是说，汽车的各个部分构成一个组，其中每一部分又都包含子部分，等等。子部分经常在它们的“本地”坐标系中定义，并可变换到它的“父空间”中，而这个“父空间”又可变换到它自己的“父空间”中，依次类推，直到这种层次的根层为止，这种根层就是一般所定义的“世界空间”。

每一层都存在一个本地坐标系。坐标系具有自己的原点，即点$[0\ \ 0\ \ 1]$或$[0\ \ 0\ \ 0\ \ 1]$，分别对应于二维和三维系统。更进一步，坐标系具有自己的名为正常基底的基底向量集（或者更正式的叫法是标准欧几里得基底）。该基底是正交的，符合右手法则，其次序如下：向量$\vec{v}_i$的第i个元素为1，其余元素为0。一般将它们分别叫做x，y和z轴。点或向量的系数，即其坐标，也分别叫做x，y和z分量。因此，在三维系统中，方程（4.5）中的矩阵为

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

二维系统与此类似。显然，它们都是单位矩阵，因此我们可以将方程（4.4）的最后一行简单地写成

$$P=[a_1\ \ a_2\ \ \cdots\ \ a_n\ \ 1]$$

在2.9.2节中已经证明，可以从其他的（线性无关）基底构建一个正交基底；因此，我们可以将这些正常基底用于所有的本地坐标系（“坐标系统”），而不会有损其表示能力。正常基底在直观应用和计算方面具有明显的优势。本章的其余部分都假定使用正常基底。

4.3　加法、减法和乘法

在4.2节中，我们说明了如何用矩阵形式来表示点和向量。现在我们要正式地说明如何用其坐标/矩阵表示来定义点和/或向量的加法、减法和数乘。

在以前所有关于仿射空间的讨论中，我们已经做了如下设定：基底向量使用像$\vec{u}$,$\vec{v}$这样的名称，它们的坐标采用像a_1,a_2这样的名字，等等。一般地，在二维和三维空间中，基底向量通常称为x，y和z轴，坐标通常直接称为x，y和z坐标。然而，为了避免在后面

的几节中引起混淆，我们将采用经常用在微积分中的一种惯例，即将基底向量称为 i， j 和 k 。我们将像以前一样地称呼原点$\mathcal{O}$（图 4.1 和图 4.2 显示了这种表示方法）。

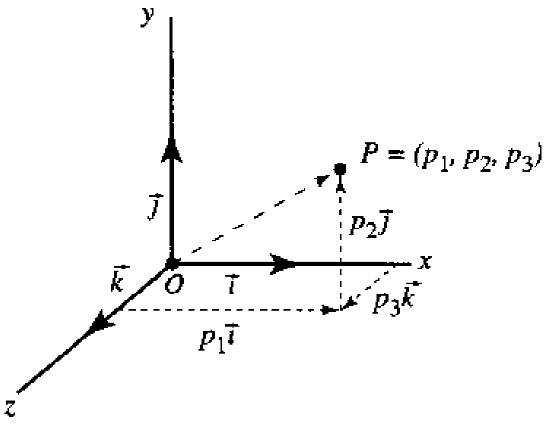


图 4.1　$P = p_1\vec{i} + p_2\vec{j} + p_3\vec{k} + \mathcal{O} = [p_1 \quad p_2 \quad p_3 \quad 1]$

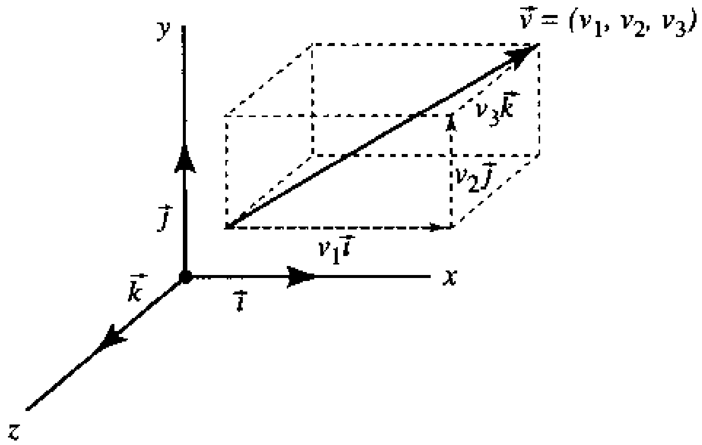


图 4.2　$\vec{v} = v_1\vec{i} + v_2\vec{j} + v_3\vec{k} = [v_1 \quad v_2 \quad v_3 \quad 0]$

4.3.1　向量加法和减法

假定有两个向量$\vec{u} = u_1\vec{i} + u_2\vec{j} + u_3\vec{k}$和$\vec{v} = v_1\vec{i} + v_2\vec{j} + v_3\vec{k}$，我们希望用它们的矩阵形式来对它们相加或相减：

$$\begin{aligned}\vec{u} + \vec{v} &= [u_1 \quad u_2 \quad u_3 \quad 0] + [v_1 \quad v_2 \quad v_3 \quad 0]\\ &= u_1\vec{i} + u_2\vec{j} + u_3\vec{k} + v_1\vec{i} + v_2\vec{j} + v_3\vec{k}\\ &= (u_1 + v_1)\vec{i} + (u_2 + v_2)\vec{j} + (u_3 + v_3)\vec{k}\\ &= [u_1 + v_1 \quad u_2 + v_2 \quad u_3 + v_3 \quad 0]\end{aligned}$$

减法的证明与此相似。

4.3.2　点与向量的加法和减法

点与向量的相加或相减类似于向量之间的加法和减法：假定有一个点$P = p_1\vec{i} + p_2\vec{j} +$

$p_3\vec{k}+\mathcal{O}$ 和一个向量$\vec{v}=v_1\vec{i}+v_2\vec{j}+v_3\vec{k}$，用矩阵来表示它们的相加：

$$\begin{aligned}P+\vec{v}&=[p_1\quad p_2\quad p_3\quad 1]+[v_1\quad v_2\quad v_3\quad 0]\\&=p_1\vec{i}+p_2\vec{j}+p_3\vec{k}+\mathcal{O}+v_1\vec{i}+v_2\vec{j}+v_3\vec{k}\\&=(p_1+v_1)\vec{i}+(p_2+v_2)\vec{j}+(p_3+v_3)\vec{k}+\mathcal{O}\\&=[p_1+v_1\quad p_2+v_2\quad p_3+v_3\quad 1]\end{aligned}$$

减法的证明与此类似。

4.3.3 点的减法

假定有两个点$P=p_1\vec{i}+p_2\vec{j}+p_3\vec{k}+\mathcal{O}$和$Q=q_1\vec{i}+q_2\vec{j}+q_3\vec{k}+\mathcal{O}$，我们希望用矩阵来表示它们的相减：

$$\begin{aligned}P-Q&=[p_1\quad p_2\quad p_3\quad 1]-[q_1\quad q_2\quad q_3\quad 1]\\&=(p_1\vec{i}+p_2\vec{j}+p_3\vec{k}+\mathcal{O})-(q_1\vec{i}+q_2\vec{j}+q_3\vec{k}+\mathcal{O})\\&=(p_1-q_1)\vec{i}+(p_2-q_2)\vec{j}+(p_3-q_3)\vec{k}\\&=[p_1-q_1\quad p_2-q_2\quad p_3-q_3\quad 0]\end{aligned}$$

4.3.4 数乘

假定有一个向量$\vec{v}=v_1\vec{i}+v_2\vec{j}+v_3\vec{k}$，我们希望将它乘以一个数$\alpha$。用矩阵表示可得

$$\begin{aligned}\alpha\vec{v}&=\alpha\,[v_1\quad v_2\quad v_3\quad 0]\\&=\alpha(v_1\vec{i}+v_2\vec{j}+v_3\vec{k})\\&=(\alpha v_1)\vec{i}+(\alpha v_2)\vec{j}+(\alpha v_3)\vec{k}\\&=[\alpha v_1\quad \alpha v_2\quad \alpha v_3\quad 0]\end{aligned}\tag{4.6}$$

4.4 向量乘积

上述关于加法、减法和乘法的证明是显而易见的并且也很简单，而在下面的几节中，我们将看到其中所采用的方法对帮助我们理解点积和叉积中的分量运算将极为有效。一般的教材在讨论这些运算时只是简单地列出公式，但我们的目的是要说明为什么这些公式是正确的。

4.4.1 点积

在 3.3.1 节中，我们抽象地、以不用坐标运算的方式讨论了两个向量的数量积（或点积）。在 2.3.4 节中，我们还讨论了$1\times n$矩阵（“行向量”）和$n\times 1$矩阵（“列向量”）的内积（特殊情形就是点积）。敏锐的读者可能已经注意到，这两种运算都叫做点积，自然，它们并不是等价的。我们明确地定义

$$\vec{u}\cdot\vec{v}=[u_1 \quad u_2 \quad \cdots \quad u_n]\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

即两个向量的数量（点）积用矩阵运算来表示就是它们的矩阵表示的内（点）积。

当然，我们感兴趣的是为什么会如此：为什么各个坐标分量相乘，然后相加，就会得到一个与向量之间的夹角相关的数值。为了理解这一点，最好采用其他途径：假定首先采用不用坐标的点积定义，再说明如何推演到矩阵的内积。这种方法与证明点和向量的加法和减法所采用的方法相似。

假设有两个向量 $\vec{u}$ 和 $\vec{v}$，由点积的定义可知：

$$\vec{u}\cdot\vec{v}=\|\vec{u}\|\|\vec{v}\|\cos\theta$$

如果将该定义应用于基底向量，可得

$$\vec{i}\cdot\vec{i}=\|\vec{i}\|\|\vec{i}\|\cos\theta=1\cdot 1\cdot 1=1$$

$$\vec{j}\cdot\vec{j}=\|\vec{j}\|\|\vec{j}\|\cos\theta=1\cdot 1\cdot 1=1$$

$$\vec{k}\cdot\vec{k}=\|\vec{k}\|\|\vec{k}\|\cos\theta=1\cdot 1\cdot 1=1$$

这是由于任何向量与其自身的夹角为 0，其余弦为 1。

成对地将点积定义应用于基底向量，则

$$\vec{i}\cdot\vec{j}=\|\vec{i}\|\|\vec{j}\|\cos\theta=1\cdot 1\cdot 0=0$$

$$\vec{i}\cdot\vec{k}=\|\vec{i}\|\|\vec{k}\|\cos\theta=1\cdot 1\cdot 0=0$$

$$\vec{j}\cdot\vec{k}=\|\vec{j}\|\|\vec{k}\|\cos\theta=1\cdot 1\cdot 0=0$$

这是由于基底向量都是单位长度向量，并且它们相互之间的夹角都为 $\frac{\pi}{2}$，其余弦为 0。

如果有向量 $\vec{u}=u_1\vec{i}+u_2\vec{j}+u_3\vec{k}$ 和 $\vec{v}=v_1\vec{i}+v_2\vec{j}+v_3\vec{k}$，则可以计算它们的点积如下

$$\begin{aligned}\vec{u}\cdot\vec{v}&=[u_1 \quad u_2 \quad u_3 \quad 0]\cdot[v_1 \quad v_2 \quad v_3 \quad 0]\\&=(u_1\vec{i}+u_2\vec{j}+u_3\vec{k})\cdot(v_1\vec{i}+v_2\vec{j}+v_3\vec{k})\\&=u_1v_1(\vec{i}\cdot\vec{i})+u_1v_2(\vec{i}\cdot\vec{j})+u_1v_3(\vec{i}\cdot\vec{k})\\&\quad+u_2v_1(\vec{j}\cdot\vec{i})+u_2v_2(\vec{j}\cdot\vec{j})+u_2v_3(\vec{j}\cdot\vec{k})\\&\quad+u_3v_1(\vec{k}\cdot\vec{i})+u_3v_2(\vec{k}\cdot\vec{j})+u_3v_3(\vec{k}\cdot\vec{k})\\&=u_1v_1+u_2v_2+u_3v_3\end{aligned}$$

4.4.2　叉积

显然用矩阵来表示点积是很麻烦的，但是用来表示叉积却并非如此。叉积的定义（参见 3.3.1 节）是相对直观的，但是该定义并没有使用单个的矩阵运算：那么，对于如下的表达式

$$\vec{w}=\vec{u}\times\vec{v} \tag{4.7}$$

如何用矩阵来实现它呢？

实际上有两种方法可以解决这一问题：

- 如果只是要计算叉积，我们如何直接进行计算？
- 我们能不能构建一个用于计算叉积的矩阵？如果有一系列涉及点积、叉积、缩放等的运算，那么，可以将该矩阵用于一系列的矩阵运算以实现这些运算。

我们将依次讨论这些方法。

1．直接计算叉积

与点积的讨论一样，我们首先回顾叉积的定义，如果有两个向量 $\vec{u}=u_1\vec{i}+u_2\vec{j}+u_3\vec{k}$ 和 $\vec{v}=v_1\vec{i}+v_2\vec{j}+v_3\vec{k}$，它们的叉积由如下三个性质定义。

i．长度：

$$\|\vec{u}\times\vec{v}\|=\|\vec{u}\|\|\vec{v}\||\sin\theta|$$

如果将其应用于每一个基底向量可得

$$\vec{i}\times\vec{i}=\vec{0}$$

$$\vec{j}\times\vec{j}=\vec{0}$$

$$\vec{k}\times\vec{k}=\vec{0}$$

这是由于任何基底向量与其自身的夹角都为 0，其正弦为 0。

ii．正交性：

$$\vec{u}\times\vec{v}\perp\vec{u}$$

$$\vec{u}\times\vec{v}\perp\vec{v}$$

iii．方向：右手法则决定了叉积的方向（3.2.4 节）。结合第二个性质，一起应用于基底向量，可得

$$\vec{i}\times\vec{j}=\vec{k}$$

$$\vec{j}\times\vec{i}=-\vec{k}$$

$$\vec{j}\times\vec{k}=\vec{i}$$

$$\vec{k}\times\vec{j}=-\vec{i}$$

$$\vec{k}\times\vec{i}=\vec{j}$$

$$\vec{i}\times\vec{k}=-\vec{j}$$

这是由于基底向量是相互垂直的并且总是遵循右手法则。

有了上述的基础，我们现在可以证明叉积公式：

$$\begin{aligned}\vec{u}\times\vec{v}&=[\,u_1\quad u_2\quad u_3\quad 0\,]\times[\,v_1\quad v_2\quad v_3\quad 0\,]\\&=(u_1\vec{i}+u_2\vec{j}+u_3\vec{k})\times(v_1\vec{i}+v_2\vec{j}+v_3\vec{k})\\&=(u_1v_1)(\vec{i}\times\vec{i})+(u_1v_2)(\vec{i}\times\vec{j})+(u_1v_3)(\vec{i}\times\vec{k})\end{aligned}$$

$$+(u_2v_1)(\vec{j}\times\vec{i})+(u_2v_2)(\vec{j}\times\vec{j})+(u_2v_3)(\vec{j}\times\vec{k})$$

$$+(u_3v_1)(\vec{k}\times\vec{i})+(u_3v_2)(\vec{k}\times\vec{j})+(u_3v_3)(\vec{k}\times\vec{k})$$

$$=(u_1v_1)\vec{0}+(u_1v_2)\vec{k}+(u_1v_3)(-\vec{j})$$

$$+(u_2v_1)(-\vec{k})+(u_2v_2)\vec{0}+(u_2v_3)\vec{i}$$

$$+(u_3v_1)\vec{j}+(u_3v_2)(-\vec{i})+(u_3v_3)\vec{0}$$

$$=(u_2v_3-u_3v_2)\vec{i}+(u_3v_1-u_1v_3)\vec{j}+(u_1v_2-u_2v_1)\vec{k}+\vec{0}$$

$$=[u_2v_3-u_3v_2\quad u_3v_1-u_1v_3\quad u_1v_2-u_2v_1\quad 0]$$

2. 用矩阵乘法实现叉积

首先举一个例子，然后再说明为什么，这可能是最好的方式。给定一个如方程 4.7 一样的表达式，按照下面的方式来观察它：

$$\vec{w}=\vec{u}\times\vec{v}$$

$$=[u_1\quad u_2\quad u_3]\begin{bmatrix}\quad ?\quad\end{bmatrix}$$

我们已知叉积的定义为

$$\vec{w}=\vec{u}\times\vec{v}$$

$$=(u_2v_3-u_3v_2,u_3v_1-u_1v_3,u_1v_2-u_2v_1)$$

根据矩阵乘法的定义，我们可反转需要求解的矩阵，一个“斜对称矩阵”，用符号$\tilde{v}$来表示：

$$\tilde{v}=\begin{bmatrix}0&-v_3&v_2\\v_3&0&-v_1\\-v_2&v_1&0\end{bmatrix}$$

综合以上各式可得：

$$\vec{w}=\vec{u}\times\vec{v}$$

$$=[u_1\quad u_2\quad u_3]\tilde{v}$$

$$=[u_1\quad u_2\quad u_3]\begin{bmatrix}0&-v_3&v_2\\v_3&0&-v_1\\-v_2&v_1&0\end{bmatrix}$$

根据不同的计算环境，我们有时可能希望反转矩阵所表示的向量。由于叉积不具备交换性，我们不能简单地交换各个u 分量和v 分量。然而，我们知道，叉积是反对称的：

$$\vec{u}\times\vec{v}=-(\vec{v}\times\vec{u})$$

而且，在 2.3.4 节中我们已经证明，可以通过转置矩阵来实现矩阵乘法次序的反转。

因此，要计算$\vec{w}=\vec{v}\times\vec{u}$（与过去一样，$\vec{u}$ 保持其一般的矩阵形式），我们有

$$\tilde{v}=\begin{bmatrix}0&v_3&-v_2\\-v_3&0&v_1\\v_2&-v_1&0\end{bmatrix}$$

结果为

$$\begin{aligned}\vec{w} &= \vec{v} \times \vec{u} \\ &= \tilde{v}\vec{u}\end{aligned}$$

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} = \begin{bmatrix} 0 & v_3 & -v_2 \\ -v_3 & 0 & v_1 \\ v_2 & -v_1 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$

4.4.3 张量积

向量代数中常见的另一个表达式是：

$$\vec{t} = (\vec{u} \cdot \vec{v})\,\vec{w} \tag{4.8}$$

为了计算下式，我们用关于$\vec{u}$的矩阵算术来表示该式

$$[t_1 \quad t_2 \quad t_3] = [u_1 \quad u_2 \quad u_3][\quad ? \quad]$$

回顾 4.4.1 节中点积的定义（产生一个数量），以及 4.3.4 节中向量的数乘的定义（产生一个向量）。我们可用它们来反转相关的矩阵。两个向量的张量积记为$\vec{v} \otimes \vec{w}$。据此可得

$$\begin{aligned}\vec{t} &= (\vec{u} \cdot \vec{v})\,\vec{w} \\ &= [u_1 \quad u_2 \quad u_3] \begin{bmatrix} v_1w_1 & v_1w_2 & v_1w_3 \\ v_2w_1 & v_2w_2 & v_2w_3 \\ v_3w_1 & v_3w_2 & v_3w_3 \end{bmatrix}\end{aligned}$$

如果将它计算出来，就可以看出这种运算其实与方程（4.8）所表示的一样。这同时也揭示了这种运算的本质，它将向量$\vec{u}$变换为一个与$\vec{w}$平行的向量：

$$\vec{t} = [\,(u_1v_1 + u_2v_2 + u_3v_3)w_1 \quad (u_1v_1 + u_2v_2 + u_3v_3)w_2 \quad (u_1v_1 + u_2v_2 + u_3v_3)w_3\,]$$

这种运算是$\vec{u}$对$\vec{v}$和$\vec{w}$的线性变换，因为它将向量变换为向量并保持线性组合：在 4.7 节中我们将见到这种运算的用处。注意向量的次序也是很重要的：一般地，$(\vec{w} \otimes \vec{v})^{\mathrm{T}} = \vec{v} \otimes \vec{w}$。

4.4.4 正交运算符和正交点积

正交点积是一种向量运算，它具有出人意料的的用处，但是可能并未得到充分的利用。在本节中，我们首先将介绍正变运算及其性质，然后说明如何用它来定义正交点积，并介绍正交点积的性质。

1. 垂直运算符

我们已在前面使用过运算符“⊥”（读做“perp”），但并没有做过多的解释。如果已有向量$\vec{v}$，那么$\vec{v}^{\perp}$就是与向量$\vec{v}$垂直的向量（参见图 4.3）。自然，在二维空间中，确实存在两个相互垂直的向量（具有相同的长度），一个位于顺时针 90°，另一个位于逆时针 90°。然而，由于我们采用右手法则，因此选择位于逆时针 90°的垂直向量更合适，如图 4.3 所示。

垂直向量频繁地出现在二维几何算法中，因此采用这种常用的表示法是很有用的。用向量来表示这种运算是很直观和清晰的。但如果用矩阵来表示又会如何呢？图 4.3 中的向

量 $\vec{v}$ 近似于[1 0.2]。直观的做法是，交换向量的两个分量，然后对第一个分量取反。用矩阵来表示，可得

$$\vec{v}^{\perp} = [1 \quad 0.2]\begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

$$= [-0.2 \quad 1]$$

在三维空间中，存在无数个与给定的向量垂直并且具有相同长度的向量（它们定义了一个与该向量垂直的“圆盘”）。不可能在三维空间中对所有的向量定义一种一致的法则以产生一个可惟一识别的“垂直”，这似乎限制了垂直运算在三维空间中的应用。因此，在本节随后的部分，我们将集中讨论二维空间的情形。

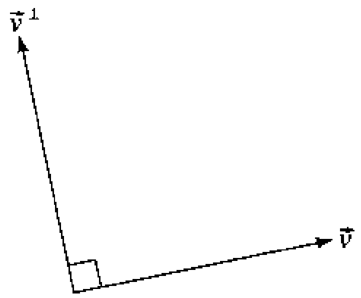


图 4.3 垂直运算

2. 性质

Hill（1994）给出了垂直运算的一些有用的性质：

i. $\vec{v}^{\perp} \perp \vec{v}$

ii. 线性

  a. $(\vec{u} + \vec{v})^{\perp} = \vec{u}^{\perp} + \vec{v}^{\perp}$

  b. $(k\vec{v})^{\perp} = k(\vec{v}^{\perp}), \forall k \in \mathbb{R}$

iii. $\|\vec{v}^{\perp}\| = \|\vec{v}\|$

iv. $\vec{v}^{\perp\perp} = (\vec{v}^{\perp})^{\perp} = -\vec{v}$

v. $\vec{v}^{\perp}$是由 $\vec{v}$ 逆时针旋转 90°所得的向量

【证明】 i. $\vec{v}^{\perp} \cdot \vec{v} = [-v_y \quad v_x] \cdot [v_x \quad v_y] = -v_y \cdot v_x + v_x \cdot v_y = 0$。由于点积为零，因此这两个向量相互垂直。

ii. a.

$$(\vec{u} + \vec{v})^{\perp} = \vec{u}^{\perp} + \vec{v}^{\perp}$$

$$([u_x \quad u_y] + [v_x \quad v_y])^{\perp} = [-u_y \quad u_x] + [-v_y \quad v_x]$$

$$[u_x + v_x \quad u_y + v_y]^{\perp} = [-(u_y + v_y) \quad u_x + v_x]$$

$$[-(u_y + v_y) \quad u_x + v_x] = [-(u_y + v_y) \quad u_x + v_x]$$

b.

$$(k\vec{v})^{\perp} = k(\vec{v}^{\perp})$$

$$(k[v_x \quad v_y])^{\perp} = k[v_x \quad v_y]^{\perp}$$

$$[kv_x \quad kv_y]^{\perp} = k[-v_y \quad v_x]$$

$$[-kv_y \quad kv_x] = [-kv_y \quad kv_x]$$

iii.
$$\sqrt{(-v_y)^2 + (v_x)^2} = \sqrt{(v_x)^2 + (v_y)^2}$$

iv.
$$\begin{array}{ccccc} \vec{v}^{\perp\perp} & = & (\vec{v}^{\perp})^{\perp} & = & -\vec{v} \\ [v_x \quad v_y]^{\perp\perp} & = & ([v_x \quad v_y]^{\perp})^{\perp} & = & -[v_x \quad v_y] \\ [-v_y \quad v_x]^{\perp} & = & [-v_y \quad v_x]^{\perp} & = & [-v_x \quad -v_y] \\ [-v_x \quad -v_y] & = & [-v_x \quad -v_y] & = & [-v_x \quad -v_y] \end{array}$$

v. 如果有一个复数$x_a + y_a \cdot i$，并乘以复数 i，可得到一个由复数$a: -y_a + x_a \cdot i$. A逆时针旋转 90°而成的复数。一个向量$\vec{v}$可看成复平面上的一个点$v_x + v_y \cdot i$，那么$\vec{v}^{\perp}$可看成点$-v_y + v_x \cdot i$。■

3. 垂直点运算

Hill 的杰出论文提供了垂直点运算的许多应用，我们鼓励大家去研究它们，以理解这种运算是如何被广泛应用的。在此我们仅仅简单介绍这种运算及其重要性。

垂直点运算就是一般的两个向量的点积的应用，其中第一个向量已被“垂直”：$\vec{u}^{\perp} \cdot \vec{v}$。在明确和证明垂直点运算的不同性质之前，我们先分析其几何性质。

（1）几何解释

有两个重要的几何性质值得考虑。我们首先回顾两个向量的标准点积与它们之间的夹角之间的关系。给定两个向量$\vec{u}$ 和$\vec{v}$，我们有

$$\cos\theta = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|} \tag{4.9}$$

（参见 3.3.1 节）。因此，如果基于它们之间的夹角（如图 4.4 所示）来考虑$\vec{u}^{\perp}$和$\vec{v}$之间的关系，可以看出

$$\cos\phi = \frac{\vec{u}^{\perp} \cdot \vec{v}}{\|\vec{u}^{\perp}\|\|\vec{v}\|}$$

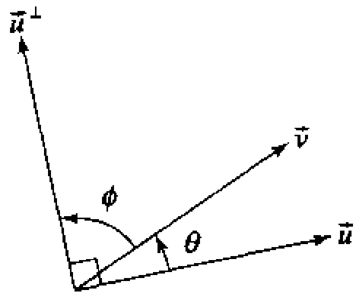


图 4.4　垂直点积反映了向量之间的有符号角度

在前面我们已经证明了$\|\vec{u}^{\perp}\| = \|\vec{u}\|$，因此上式可以改写为

$$\cos\phi = \frac{\vec{u}^{\perp} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|} \tag{4.10}$$

或

$$\vec{u}^{\perp} \cdot \vec{v} = \|\vec{u}\|\|\vec{v}\| \cos\phi \tag{4.11}$$

为了进一步推导，我们注意到，如果$\theta+\phi=\pi/2$，则$\sin\theta=\cos\phi$（我们鼓励你采用所拥有的数学知识来校验这一点）。从图中可以直接看出，确实$\theta+\phi=\pi/2$，因此方程（4.10）可改写为

$$\vec{u}^{\perp}\cdot\vec{v}=\|\vec{u}\|\|\vec{v}\|\sin\theta$$

如果假设两个向量是规范的，就可能有助于我们的推导。这种情形下，有

$$\hat{u}^{\perp}\cdot\hat{v}=\sin\theta \tag{4.12}$$

这就非常清楚地说明，垂直点积运算不仅反映了两个向量之间的夹角，而且指出了它们之间的夹角的方向（即符号）。与两个向量的（一般）点积与它们之间角度的关系（如方程（4.9）所示）相比，一般点积仅仅指明了两个向量之间的角度，但不能说明角度的方向（即角度的符号）。

垂直点积运算的第二个几何性质可以从图 4.5 中看出。根据定义，$\sin\theta=h/\|\vec{v}\|$，因此$\vec{u}$和$\vec{v}$所定义的平行四边形的高度为$h=\|\vec{v}\|\sin\theta$。其底自然是$\|\vec{u}\|$，因此我们有

$$\text{Area}=\|\vec{u}\|\|\vec{v}\|\sin\theta$$

注意，方程的右边与方程（4.12）相同，从中可得出如下结论：两个向量的垂直点积运算等于这两个向量所定义的三角形的有符号面积的两倍。从 3.3.1 节中可知，两个向量的叉积（在三维空间中）与平行四边形的面积相关，因此我们现在可以看到，正如 Hill（1994）所指出的，垂直点积可以被看成是三维叉积的二维模拟。

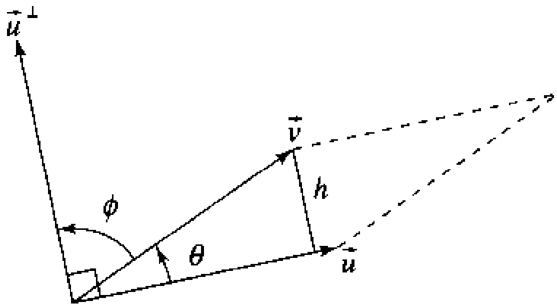


图 4.5　垂直点积与两个向量所构成的三角形的有符号面积相关

说明这一点的另一种方法是简单地用其分量来表示垂直点积：

$$\begin{aligned}\vec{u}^{\perp}\cdot\vec{v}&=-u_yv_x+u_xv_y\\&=u_xv_y-u_yv_x\\&=\begin{vmatrix}u_x & u_y\\ v_x & v_y\end{vmatrix}\end{aligned}$$

如果我们将$\vec{u}$和$\vec{v}$看成是嵌入平面$z=0$中的三维向量，那么可以看出，上式就是三元数积$(\vec{u}\times\vec{v})\cdot[0\quad 0\quad 1]$。

正如 3.3.1 节中的方程 3.13 所示，定义了平行六面体的体积的矩阵以三个向量作为行，多元组数量积与该矩阵的行列式相关。在我们讨论的情形中，高度为 1，因此该矩阵也与面积相关。

（2）性质

我们列举几个垂直点积与一般点积不同的性质作为本节的结语：

i．$\vec{u}^{\perp}\cdot\vec{v}=-\vec{v}^{\perp}\cdot\vec{u}$。一般点积的规则是$\vec{u}\cdot\vec{v}=\vec{v}\cdot\vec{u}$。

ii．$\vec{v}^{\perp}\cdot\vec{v}=0$。一般点积的规则是$\vec{v}\cdot\vec{v}=\|\vec{v}\|^2$。

iii．$\vec{u}^{\perp}\cdot\vec{v}=\|\vec{u}\|\|\vec{v}\|\sin\theta$。一般点积的规则是$\vec{u}\cdot\vec{v}=\|\vec{u}\|\|\vec{v}\|\cos\theta$。

4.5 仿射变换的矩阵表示

在 2.1.1 节中，我们已经说明，矩阵乘法按照如下几种不同的方式来进行解释：坐标变化、平面内的变换或者从一个平面到另一个平面的变换。在 4.6 节中，我们将讨论建立一个进行基底变换的矩阵的一般方法。在本节中，我们讨论如何建立一个对点和向量进行仿射变换的矩阵。

假定我们有两个仿射空间$\mathcal{A}$和$\mathcal{B}$，我们为它们分别任意选择一个坐标系$\mathcal{F}_{\mathcal{A}}(\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n,\mathcal{O}_{\mathcal{A}})$和$\mathcal{F}_{\mathcal{B}}(\vec{w}_1,\vec{w}_2,\ldots,\vec{w}_n,\mathcal{O}_{\mathcal{B}})$。如果我们任意选择点$P$，那么可以用$(\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n,\mathcal{O}_{\mathcal{A}})$将其位置描述为$[a_1\quad a_2\quad\ldots\quad a_n\quad 1]$。如果我们有将$\mathcal{A}$映射到$\mathcal{B}$的仿射变换$T$，那么我们用符号$T(P)$来表述这个变换对点$P$进行变换而得到的点。这很抽象并且与坐标无关，但由于我们现在讨论矩阵表示（在这里，就是坐标），因此我们希望能找到点$T(P)$所对应的坐标$(\vec{w}_1,\vec{w}_2,\ldots,\vec{w}_n,\mathcal{O}_{\mathcal{B}})$。

我们在 4.2 节中已经说明了如何用矩阵来表示点（或向量），因此我们可以将$T(P)$扩展为

$$T\left(a_1\vec{v}_1+a_2\vec{v}_2+\cdots+a_n\vec{v}_n+\mathcal{O}_{\mathcal{A}}\right)\tag{4.13}$$

我们可以引用保持仿射组合的性质（方程（3.14））来将方程（4.13）改写为

$$a_1T\left(\vec{v}_1\right)+a_2T\left(\vec{v}_2\right)+\cdots+a_nT\left(\vec{v}_n\right)+T\left(\mathcal{O}_{\mathcal{A}}\right)$$

实体$T(\vec{v}_i)$自然就是更多的向量，而$T(\mathcal{O}_{\mathcal{A}})$就是一个点（因为，根据定义，像$T$这样的仿射变换将向量映射到向量，将点映射到点）。因此，它们有一种相对于$(\vec{w}_1,\vec{w}_2,\ldots,\vec{w}_n,\mathcal{O}_{\mathcal{B}})$的表示$(c_1,c_2,\cdots,c_n)$（即坐标）。我们可以将它们表示为

$$\begin{aligned}
T\left(\vec{v}_1\right)&=c_{1,1}\vec{w}_1+c_{1,2}\vec{w}_2+\cdots+c_{1,n}\vec{w}_n\\
T\left(\vec{v}_2\right)&=c_{2,1}\vec{w}_1+c_{2,2}\vec{w}_2+\cdots+c_{2,n}\vec{w}_n\\
&\vdots\\
T\left(\vec{v}_n\right)&=c_{n,1}\vec{w}_1+c_{n,2}\vec{w}_2+\cdots+c_{n,n}\vec{w}_n\\
T\left(\mathcal{O}_{\mathcal{A}}\right)&=c_{n+1,1}\vec{w}_1+c_{n+1,2}\vec{w}_2+\cdots+c_{n+1,n}\vec{w}_n+\mathcal{O}_{\mathcal{B}}
\end{aligned}$$

有了这些结论，我们现在就能很方便地建立表示变换$T:\mathcal{A}\longrightarrow\mathcal{B}$的矩阵$\mathbf{T}$

$$
T(P)=[a_1\ \ a_2\ \ \cdots\ \ a_n\ \ 1]\begin{bmatrix} T(\vec{v}_1)\\ T(\vec{v}_2)\\ \vdots \\ T(\vec{v}_n)\\ T(\mathcal{O}_{\mathcal{A}}) \end{bmatrix}
$$

$$
=[a_1\ \ a_2\ \ \cdots\ \ a_n\ \ 1]\begin{bmatrix} c_{1,1}\vec{w}_1+c_{1,2}\vec{w}_2+\cdots+c_{1,n}\vec{w}_n\\ c_{2,1}\vec{w}_1+c_{2,2}\vec{w}_2+\cdots+c_{2,n}\vec{w}_n\\ \vdots \\ c_{n,1}\vec{w}_1+c_{n,2}\vec{w}_2+\cdots+c_{n,n}\vec{w}_n\\ c_{n+1,1}\vec{w}_1+c_{n+1,2}\vec{w}_2+\cdots+c_{n+1,n}\vec{w}_n+\mathcal{O}_{\mathcal{B}} \end{bmatrix} \tag{4.14}
$$

$$
=[a_1\ \ a_2\ \ \cdots\ \ a_n\ \ 1]\begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} & 0\\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} & 0\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} & 0\\ c_{n+1,1} & c_{n+1,2} & \cdots & c_{n+1,n} & 1 \end{bmatrix}\begin{bmatrix} \vec{w}_1\\ \vec{w}_2\\ \vdots\\ \vec{w}_n\\ \mathcal{O}_{\mathcal{B}} \end{bmatrix}
$$

显然，方程 4.14 最后一行的最右边的矩阵就是$\mathcal{B}$的坐标系，而且

$$
[a_1\ \ a_2\ \ \cdots\ \ a_n\ \ 1]\begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} & 0\\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} & 0\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} & 0\\ c_{n+1,1} & c_{n+1,2} & \cdots & c_{n+1,n} & 1 \end{bmatrix} \tag{4.15}
$$

$$
=[a_1\ \ a_2\ \ \cdots\ \ a_n\ \ 1]\mathrm{T}
$$

就是一个点，这可以通过矩阵乘法的定义（参见 2.3.4 节）来证明。综上所述，矩阵乘积定义了$\mathcal{B}$中的一个点，即$T(P)$，其坐标是方程（4.15）中的矩阵相乘所得结果的行矩阵的元素。我们把这个$(n+1)\times(n+1)$的矩阵T称为变换T的矩阵表示。

注意，T的第一个n行就是$\mathcal{A}$的变换后的基底向量，最后一行就是变换后的原点。在 3.4 节的最后，我们已经说明仿射变换完全由其对n个基底向量的运算所确定。观察得到的结果，即表示变换的矩阵由基底向量矩阵表示的坐标变换所定义，这就是上述事实的矩阵表现形式。

4.6 基底变化/帧/坐标系统

在 3.2.5 节中已经说明，点或向量可以在不同的参考坐标系中表示。换句话说，如果我们在某一个空间中有一个固定点，那么我们可选择任何任意的坐标系并可惟一地确定该点相对于这个坐标系的坐标。回忆一下，这一结论的计算非常繁琐。现在我们将说明矩阵是如何方便地应用于改变基底的变换的。此外，从向量代数的观点来看，这种矩阵的建立也是很直观的。

在上一节中，我们已经说明可利用矩阵将点$P=(a_1,a_2,\ldots,a_n,1)$表示为与仿射坐标系

$\mathcal{F}_{\mathcal{A}}=(\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n,\mathcal{O}_{\mathcal{A}})^{\mathrm{T}}$相关的形式。如果我们有另一个坐标系$\mathcal{F}_{\mathcal{G}}=(\vec{w}_1,\vec{w}_2,\ldots,\vec{w}_n,\mathcal{O}_{\mathcal{G}})^{\mathrm{T}}$，那么该如何计算$P$相对于这些基底向量和原点的坐标呢？（参见图 4.6）。在 3.2.5 节中，我们已经介绍了一种计算方法，现在我们来说明使用矩阵的方法。

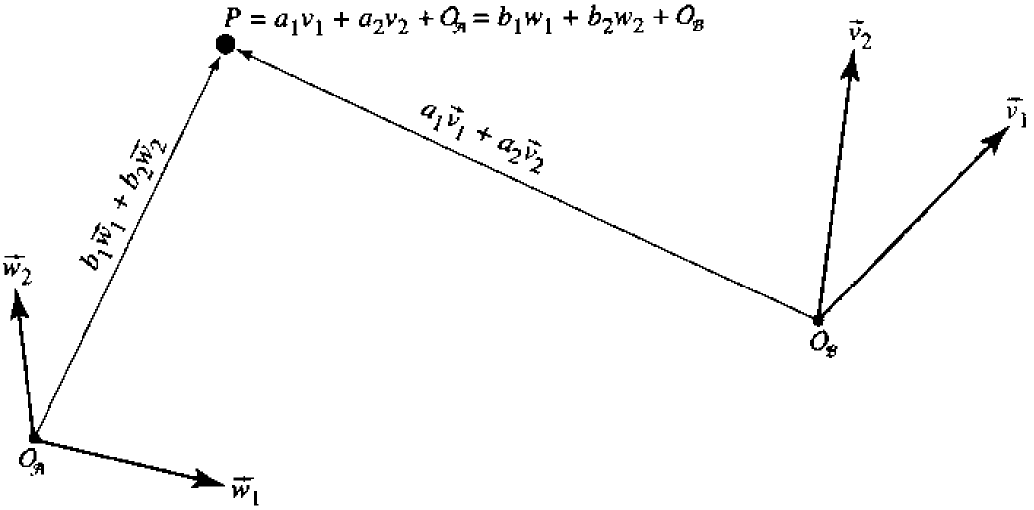


图 4.6　在$\mathcal{A}$和$\mathcal{B}$中表示P

上一节说明了用矩阵来将点表示为行矩阵（包括其坐标）与一个$(n+1)\times n$的矩阵（包括坐标系的基底向量和原点）的乘法的方法。那么，如果我们有另一个坐标系的另一组基底向量和原点，则这个问题演变为计算P 的坐标的行矩阵：

$$P=\begin{bmatrix}a_1 & a_2 & \cdots & a_n & 1\end{bmatrix}\begin{bmatrix}\vec{v}_1\\ \vec{v}_2\\ \vdots\\ \vec{v}_n\\ \mathcal{O}_{\mathcal{A}}\end{bmatrix}=\begin{bmatrix}b_1 & b_2 & \cdots & b_n & 1\end{bmatrix}\begin{bmatrix}\vec{w}_1\\ \vec{w}_2\\ \vdots\\ \vec{w}_n\\ \mathcal{O}_{\mathcal{B}}\end{bmatrix}\tag{4.16}$$

如果我们在方程的两边扩展矩阵，可得

$$\begin{aligned}&\begin{bmatrix}a_1 & a_2 & \cdots & a_n & 1\end{bmatrix}\begin{bmatrix}v_{1,1} & v_{1,2} & \cdots & v_{1,n} & 0\\ v_{2,1} & v_{2,2} & \cdots & v_{2,n} & 0\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ v_{n,1} & v_{n,2} & \cdots & v_{n,n} & 0\\ \mathcal{O}_{\mathcal{A},1} & \mathcal{O}_{\mathcal{A},2} & \cdots & \mathcal{O}_{\mathcal{A},n} & 1\end{bmatrix}\\ &=\begin{bmatrix}b_1 & b_2 & \cdots & b_n & 1\end{bmatrix}\begin{bmatrix}w_{1,1} & w_{1,2} & \cdots & w_{1,n} & 0\\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & 0\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ w_{n,1} & w_{n,2} & \cdots & w_{n,n} & 0\\ \mathcal{O}_{\mathcal{B},1} & \mathcal{O}_{\mathcal{B},2} & \cdots & \mathcal{O}_{\mathcal{B},n} & 1\end{bmatrix}\end{aligned}\tag{4.17}$$

在 3.2.3 节中，我们说明了任何点（向量）如何表示为一个基底向量的惟一的仿射组合。这里，我们可以将此定理应用于向量$(\vec{v}_1,\vec{v}_2,\ldots,\vec{v}_n)$和点$\mathcal{O}_{\mathcal{A}}$：

$$\vec{v}_1=c_{1,1}\vec{w}_1+c_{1,2}\vec{w}_2+\cdots+c_{1,n}\vec{w}_n\tag{4.18}$$

$$\vec{v}_2=c_{2,1}\vec{w}_1+c_{2,2}\vec{w}_2+\cdots+c_{2,n}\vec{w}_n\tag{4.19}$$

$$\vdots$$

$$\vec{v}_n = c_{n,1}\vec{w}_1 + c_{n,2}\vec{w}_2 + \cdots + c_{n,n}\vec{w}_n \tag{4.20}$$

$$\mathcal{O}_{\mathcal{A}} = c_{n+1,1}\vec{w}_1 + c_{n+1,2}\vec{w}_2 + \cdots + c_{n+1,n}\vec{w}_n + 1 \cdot \mathcal{O}_{\mathcal{B}} \tag{4.21}$$

换句话说，$\vec{c}_i$ 是相对于仿射坐标系$(\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n, \mathcal{O}_{\mathcal{B}})$的$\vec{v}_i$和$\mathcal{O}_{\mathcal{A}}$的坐标（分别参见图 4.7 和图 4.8）。

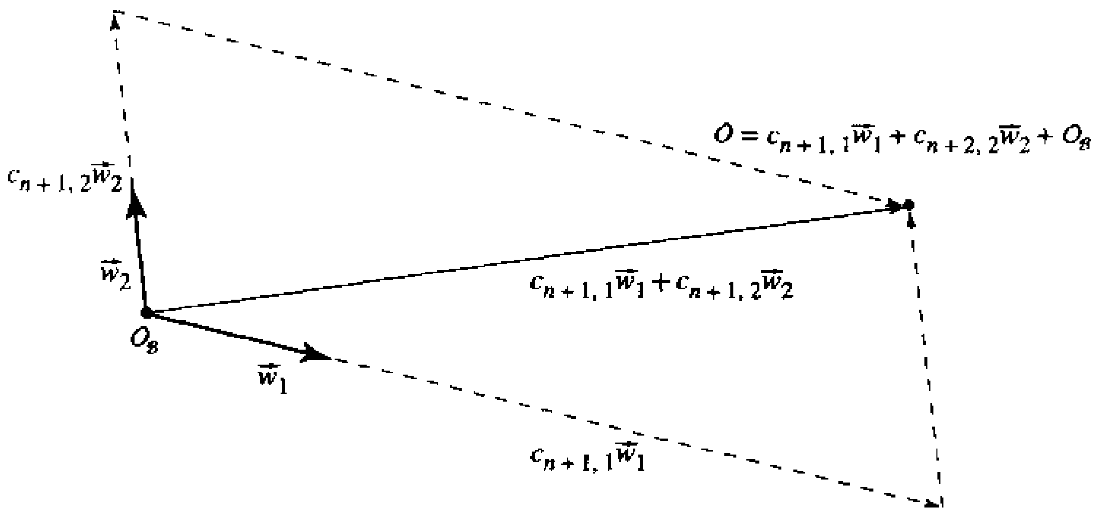


图 4.7　在$\mathcal{G}$中表示$\mathcal{O}$

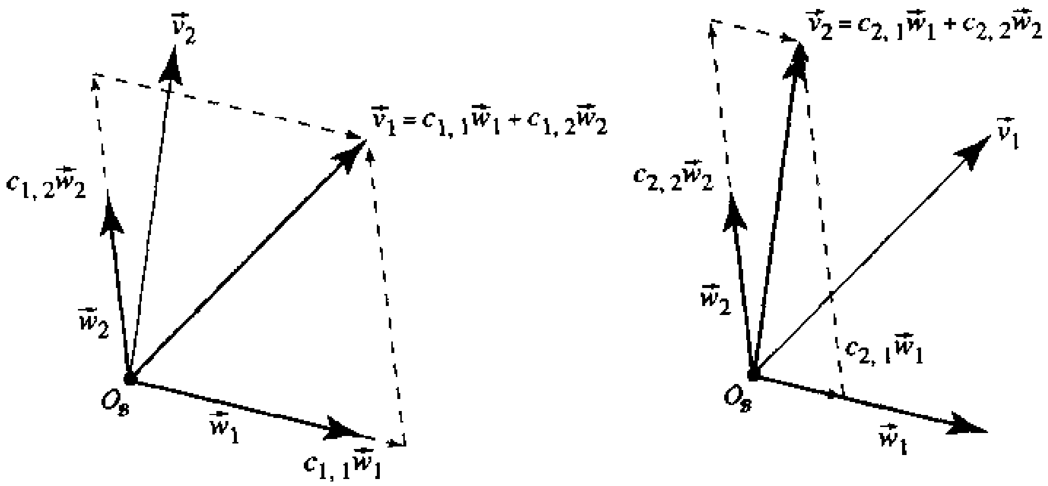


图 4.8　在$\mathcal{B}$中表示$\vec{v}_i$

我们也可以用矩阵表示来说明这一点。我们首先说明可以将方程（4.18）到方程（4.20）和方程（4.21）的右式写成矩阵形式：

$$\begin{bmatrix} c_{1,1}\vec{w}_1 + c_{1,2}\vec{w}_2 + \cdots + c_{1,n}\vec{w}_n \\ c_{2,1}\vec{w}_1 + c_{2,2}\vec{w}_2 + \cdots + c_{2,n}\vec{w}_n \\ \vdots \\ c_{n,1}\vec{w}_1 + c_{n,2}\vec{w}_2 + \cdots + c_{n,n}\vec{w}_n \\ c_{n+1,1}\vec{w}_1 + c_{n+1,2}\vec{w}_2 + \cdots + c_{n+1,n}\vec{w}_n + \mathcal{O}_{\mathcal{B}} \end{bmatrix} \tag{4.22}$$

如果参照矩阵乘法的定义（2.3.4 节），我们发现可以将方程（4.22）改写为两个分离的矩阵：

$$\begin{bmatrix} c_{1,1}\vec{w}_1 + c_{1,2}\vec{w}_2 + \cdots + c_{1,n}\vec{w}_n \\ c_{2,1}\vec{w}_1 + c_{2,2}\vec{w}_2 + \cdots + c_{2,n}\vec{w}_n \\ \vdots \\ c_{n,1}\vec{w}_1 + c_{n,2}\vec{w}_2 + \cdots + c_{n,n}\vec{w}_n \\ c_{n+1,1}\vec{w}_1 + c_{n+1,2}\vec{w}_2 + \cdots + c_{n+1,n}\vec{w}_n + \mathcal{O}_{\mathcal{B}} \end{bmatrix} = \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} & 0 \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} & 0 \\ c_{n+1,1} & c_{n+1,2} & \cdots & c_{n+1,n} & 1 \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & 0 \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,n} & 0 \\ \mathcal{O}_{\mathcal{B},1} & \mathcal{O}_{\mathcal{B},2} & \cdots & \mathcal{O}_{\mathcal{B},1} & 1 \end{bmatrix} \tag{4.23}$$

因此可以将方程（4.16）改写为

$$\begin{aligned} &[a_1 \quad a_2 \quad \cdots \quad a_n \quad 1] \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} & 0 \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} & 0 \\ c_{n+1,1} & c_{n+1,2} & \cdots & c_{n+1,n} & 1 \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & 0 \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,n} & 0 \\ \mathcal{O}_{\mathcal{B},1} & \mathcal{O}_{\mathcal{B},2} & \cdots & \mathcal{O}_{\mathcal{B},n} & 1 \end{bmatrix} \\ &= [b_1 \quad b_2 \quad \cdots \quad b_n \quad 1] \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} & 0 \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ w_{n,1} & w_{n,2} & \cdots & w_{n,n} & 0 \\ \mathcal{O}_{\mathcal{B},1} & \mathcal{O}_{\mathcal{B},2} & \cdots & \mathcal{O}_{\mathcal{B},n} & 1 \end{bmatrix} \end{aligned} \tag{4.24}$$

如果在方程的两边消去公用的矩阵因子，可得

$$\begin{aligned} &[a_1 \quad a_2 \quad \cdots \quad a_n \quad 1] \begin{bmatrix} c_{1,1} & c_{1,2} & \cdots & c_{1,n} & 0 \\ c_{2,1} & c_{2,2} & \cdots & c_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,n} & 0 \\ c_{n+1,1} & c_{n+1,2} & \cdots & c_{n+1,n} & 1 \end{bmatrix} \\ &= [b_1 \quad b_2 \quad \cdots \quad b_n \quad 1] \end{aligned}$$

4.7 向量几何和仿射变换

上一节已经提到，我们可以通过将表示变换基底向量和原点的行矩阵简单地“加在一

起”来建立变换T的矩阵T。这非常有趣和简洁，但在实际应用中，我们必须了解怎样才能为每一个仿射变换的基础类型建立相应的矩阵。下面的几节用向量代数（不用坐标）的方法描述了这些问题，并说明了如何将这种方法翻译（暂且这样说）成矩阵表示。就本质而言，我们需要做的就是弄清楚矩阵如何操作原点和仿射空间的基底向量（实际上，可以说是任意的线性无关的向量集）。

对于相对简单的变换来说，这种建立方法产生的矩阵类似于一些常规方法（即平移和均衡缩放）所生成的矩阵。其他的变换则不同，因为我们将用更一般的基于向量代数的方法来处理它们。例如，常规的旋转方法说明的就是如何建立一个旋转基底向量（坐标轴）的矩阵，以及如何建立和连接这样的一些矩阵，以实现一个相对于任意轴的变换。但在我们的处理方法中，我们说明了如何直接建立一般的旋转矩阵。为了说明常规矩阵其实就是一般方法的简单子集，我们说明了我们的方法对这些受限的子集所做的工作。例如，我们还说明了如果将旋转轴限制为坐标轴，我们的相对任意轴的旋转矩阵是如何建立常规矩阵的。

4.7.1　标记法

现在来说明如何用向量代数的方法来为仿射矩阵建立矩阵。在这一过程中我们将发现，采用一种更“示意性的”方法来表示矩阵的内容是非常方便的。你应该知道一个$(n+1)\times(n+1)$的仿射矩阵T可被分解为三部分：

1. $n\times n$的左上角子矩阵**A**
2. $(n+1)\times 1$的右列，其形式为$[0\quad 0\quad \cdots\quad 1]^{\mathrm{T}}$
3. $1\times n$的底行，其形式总是为$\vec{b}=[b_1\quad b_2\quad \cdots\quad b_n]$

这种分隔方式可描述为

$$\mathrm{T}=\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & 0 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & 0 \\ b_1 & b_2 & \cdots & b_n & 1 \end{bmatrix}$$

$$=\begin{bmatrix} \mathbf{A} & \vec{0}^{\mathrm{T}} \\ \vec{b} & 1 \end{bmatrix}$$

一般将它称为分块矩阵。

4.7.2　平移

图 4.9 显示了一个平移变换，P 是一个任意点，$\vec{w}$ 是任意向量，$\vec{u}$ 是平移向量，其坐标为$[u_1\quad u_2\quad \ldots\quad u_n\quad 0]$。仿射坐标系由基底向量$\vec{v}_i$和原点$\mathcal{O}$定义。设$Q$ 使$Q-P=\vec{w}$成立。我们来看看P是如何被平移的。显然：

$$T(P)=P+\vec{u}$$

为了得到变换矩阵的最后一行，我们需要理解原点是如何被平移的。如果我们对$\mathcal{O}$做变换T，则可得

$$
\begin{aligned}
T(\mathcal{O}) &= \mathcal{O} + \vec{w} \\
&= \vec{w} + \mathcal{O} \\
&= (u_1\vec{v}_1 + u_2\vec{v}_2 + \cdots + u_n\vec{v}_n) + \mathcal{O}
\end{aligned}
$$

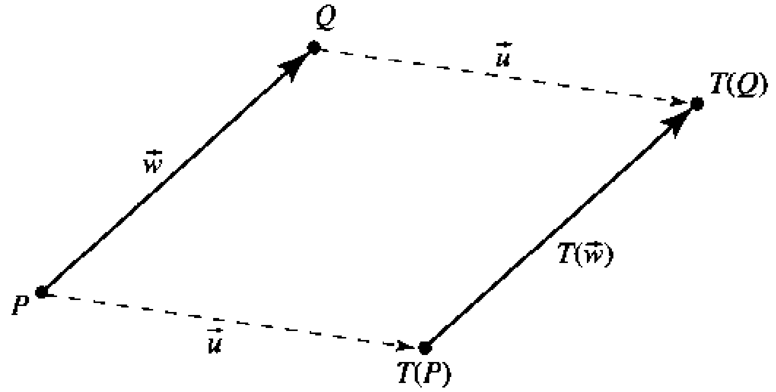


图 4.9　平移

那么，平移变换是如何影响向量的呢？我们还有如下的等式：

$$T(Q) = Q + \vec{u}$$

由于点P和Q被同一向量所平移，而且向量显然平行于其自身，因此我们可以很直观地得到，$T(\vec{w})$也平行于$\vec{w}$，并与之具有相同的长度。根据向量相等的定义（参见 3.1.1 节），可得

$$T(\vec{w}) = \vec{w}$$

当然，这就是我们所期望的：向量被定义为与位置无关的（它们仅定义了一个方向），因此平移它们并不会改变它们。可正式将其表示为：

$$
\begin{aligned}
T(\vec{w}) &= T(Q-P) && \text{点的减法定义} \\
&= T(Q) - T(P) && \text{T 的定义} \\
&= (P+\vec{w}) - (P+\vec{w}) && \text{向量和点的平移定义} \\
&= P - P && \text{方程 3.3} \\
&= \vec{w}
\end{aligned}
$$

平移没有改变向量的方向和长度，当然这对基底向量也成立；因此，T的矩阵表示的前三行就是产生原始基底向量的系数。如果点P的坐标是$[p_1\ p_2\ \cdots\ p_n]$，则

$$
T(P) = [\,p_1 \quad p_2 \quad \cdots \quad p_n \quad 1\,] \begin{bmatrix} T(\vec{v}_1) \\ T(\vec{v}_2) \\ \vdots \\ T(\vec{v}_n) \\ T(\mathcal{O}) \end{bmatrix}
$$

$$
= [\,p_1 \quad p_2 \quad \cdots \quad p_n \quad 1\,] \begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vdots \\ \vec{v}_n \\ \mathcal{O}+\vec{u} \end{bmatrix}
$$

$$= [p_1 \quad p_2 \quad \cdots \quad p_n \quad 1]\mathbf{T}\begin{bmatrix}\vec{v}_1\\ \vec{v}_2\\ \vdots\\ \vec{v}_n\\ \mathcal{O}\end{bmatrix}$$

根据矩阵乘法的定义，可得

$$\mathbf{T}=\begin{bmatrix}1 & 0 & \cdots & 0 & 0\\ 0 & 1 & \cdots & 0 & 0\\ \vdots & 0 & 1 & 0 & 0\\ 0 & 0 & \cdots & 1 & 0\\ u_1 & u_2 & \cdots & u_n & 1\end{bmatrix}$$

$$=\begin{bmatrix}\mathrm{I} & \vec{0}^{\mathrm{T}}\\ \vec{u} & 1\end{bmatrix}$$

该式的意义如图 4.10 所示。

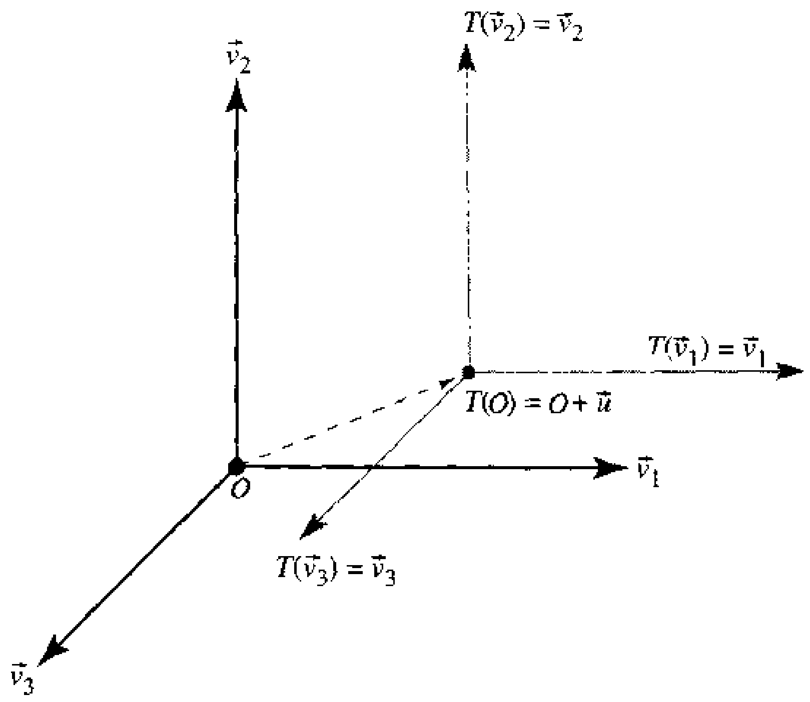


图 4.10　坐标系的平移

4.7.3　旋转

三维空间的旋转经常作为相对一个坐标轴的旋转来处理，一般的旋转往往简化为这种简单的情形（用平移来实现）。我们先描述这种简单的情形，然后继续说明如何直接且更有效地解决一般情形的问题。

1．简单情形

最一般的旋转是指相对于任意中心、任意轴和角的旋转。这里先描述如下最简单的形式，再来说明一般的情形：坐标系的原点是旋转中心，而且旋转轴是坐标系的基底向量之

一（相对坐标轴的旋转）。仅仅使用向量代数原理便可以直接为其建立一个矩阵（如图 4.11 所示）。我们将描述如何为相对 z 轴且旋转角为 θ 的旋转建立矩阵**T**。对于将讨论的三维空间的情形，我们采用如下的表示法来表示矩阵的元素：

$$\mathbf{T}=\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & 0 \\ a_{2,1} & a_{2,2} & a_{2,3} & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

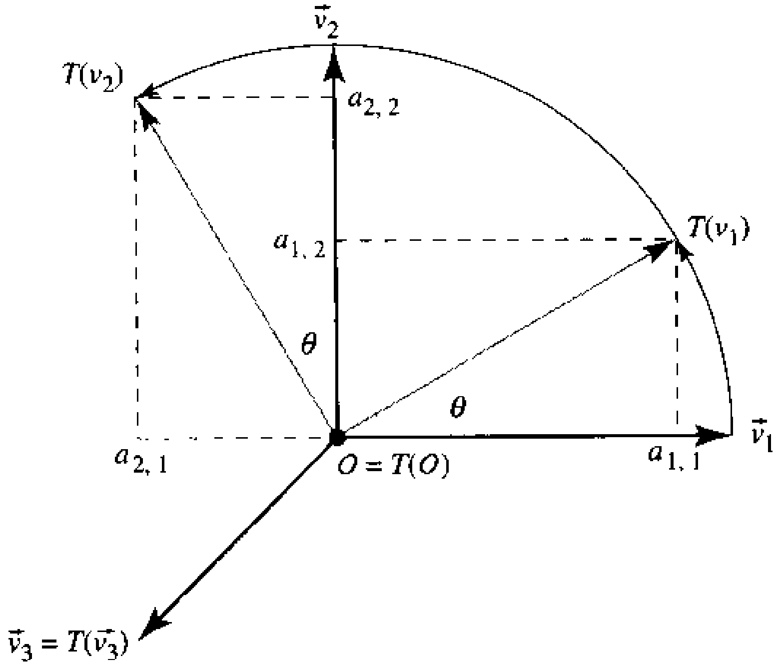


图 4.11　坐标系的简单旋转

首先考虑相对 x 轴的变换结果，即 $\vec{v}_1$。从我们关于点积的讨论中（3.3.1 节）可知，$T(\vec{v}_1)$ 可以分解为 $T(\vec{v}_1)_{\parallel}$ 和 $T(\vec{v}_1)_{\perp}$（相对于 $\vec{v}_1$）。根据点积的定义（方程（3.5），方程（3.6）和方程（3.7）），可得

$$\cos\theta=\frac{T(\vec{v}_1)\cdot\vec{v}_1}{\|T(\vec{v}_1)\|\|\vec{v}_1\|}$$

由于 $\|T(\vec{v}_1)\|=1$ 且 $\|\vec{v}_1\|=1$，因此

$$\cos\theta=T(\vec{v}_1)\cdot\vec{v}_1$$

回顾点积将 $T(\vec{v}_1)$ 投射到 $\vec{v}_1$ 上，我们可以得出如下结论：$a_{1,1}=\cos\theta$。

我们可以用相似的方法来计算 $a_{1,2}$：$T(\vec{v}_1)$ 与 $\vec{v}_2$（y 轴）之间的夹角是 $(\pi/2-\theta)$。如果我们应用刚才对 x 轴用过的推理，则可得 $a_{1,2}=\cos(\pi/2-\theta)$。根据三角几何可知 $\cos(\pi/2-\theta)=\sin\theta$，因此可以得出结论：$a_{1,2}=\sin\theta$。因此，得到基底向量 $\vec{v}_1$ 的经过变换的坐标：

$$T(\vec{v}_1)=[\,\cos\theta \quad \sin\theta \quad 0 \quad 0\,]$$

可用相同的方法来计算 $a_{2,1}$ 和 $a_{2,2}$，以及 $T(\vec{v}_2)$ 的坐标（进行旋转变换 T 后的 y 轴映像）。$T(\vec{v}_2)$ 在 $\vec{v}_1$ 上的投影是 $-\cos(\pi/2-\theta)$（符号为负是因为它指向与 $\vec{v}_1$ 相反的方向，参见方程

(3.4))，因此可得$a_{2,1} = -\sin\theta$。根据$T(\vec{v}_2)$在$\vec{v}_2$上的投影（点积）可得$a_{2,2} = \cos\theta$。因此，得到基底向量$\vec{v}_1$的经过变换的坐标：

$$T(\vec{v}_2) = [-\sin\theta \quad \cos\theta \quad 0 \quad 0]$$

其次，考虑矩阵相对z轴的变换，显然

$$T(\vec{v}_3) = \vec{v}_3$$
$$= [0 \quad 0 \quad 1 \quad 0]$$

最后，考虑矩阵相对原点的变换，显然

$$T(\mathcal{O}) = \mathcal{O}$$
$$= [0 \quad 0 \quad 0 \quad 1]$$

因此，可以简单地用基底向量和原点进行变换T后所得到的映像来建立变换的矩阵：

$$\mathbf{T}_z(\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

对于相对x轴和y轴的旋转，利用相似的推理可以得如下的简单旋转矩阵

$$\mathbf{T}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

和

$$\mathbf{T}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

显然，可以用三角几何方法来计算这些值，这也说明了基底向量是正交的这一事实。

2. 一般旋转

点相对基底向量（坐标轴）的旋转可能出现在任何的图形应用中，一般的情形是相对任意方向的轴旋转一定的角度。为了介绍对于这种问题的处理方法，大多数的图形学教材都解释如何通过将其分解成一系列的独立步骤来建立一般旋转的矩阵，即将旋转轴上的一个点平移到原点、计算相对不同坐标轴的三个不同旋转角度的行列式，并且平移并“抵消”最开始的平移。每一步的矩阵都被计算出来，并通过将所有的这些矩阵相乘而得到最终的表示一般旋转的矩阵。

这种常规方法能得到正确的结果，因此我们能够用矩阵来实现目标，但是这种方法非常复杂，其结果矩阵基本上是一个“黑匣子”，即它并没有提供理解旋转矩阵的性质和特征的信息。

在本节中，将说明如何用（不使用坐标的）向量代数来定义一般旋转，以及如何用这种方法来直接建立旋转矩阵（即不将其分解为一系列的平移和欧拉旋转）。我们希望这种方法能让你直观地理解旋转矩阵的结构和性质。简而言之，我们希望能够说明一般旋转的

工作原理，而不仅仅说明如何用特别的三角几何运算来实现旋转。

图 4.12 显示了点和向量相对一个任意轴旋转的一般情形。该图及其后的一幅图有些复杂，下面是其中用到的符号的说明：

$Q, \hat{u}$	定义旋转轴的点和单位向量
θ	旋转角度
P	被旋转的点
$T(P)$	旋转得到的点
$\vec{v}$	被旋转的向量
$T(\vec{v})$	旋转得到的向量
$\mathcal{A}$	垂直于 $\hat{u}$ 的平面
$\vec{v}_\perp$	$\vec{v}$ 在 $\mathcal{A}$ 上的投影

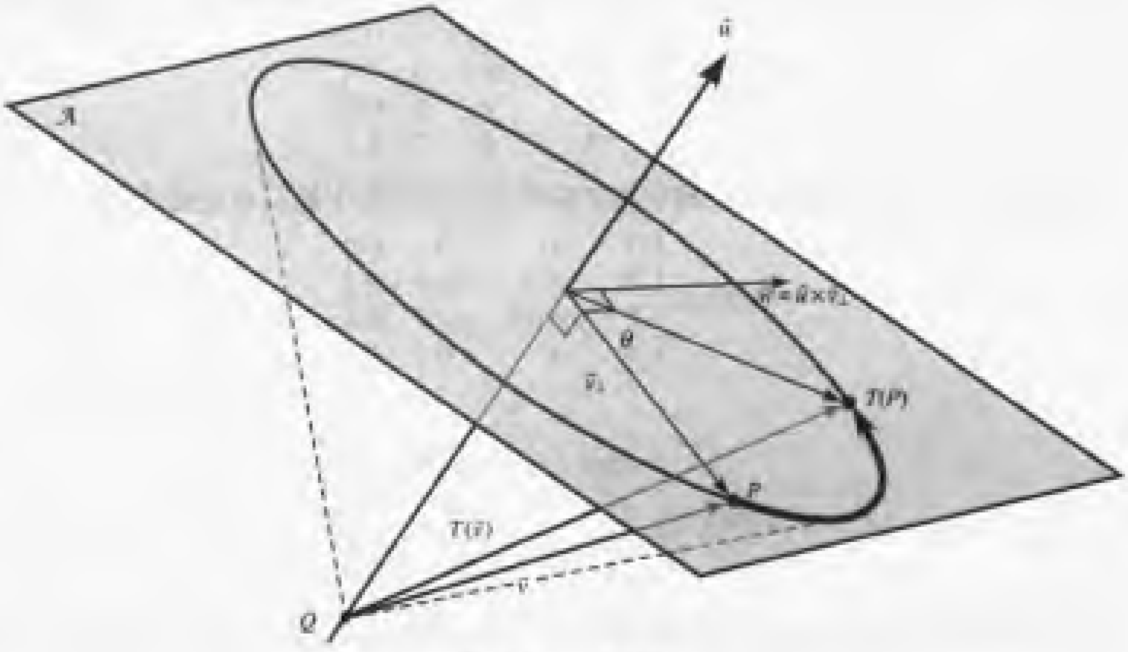


图 4.12　一般旋转

我们考虑一个相对（单位）向量 $\hat{u}$（它是经过 Q 的一条旋转轴）逆时针方向旋转并且旋转角度为 θ（右手法则）的旋转。为了讨论的方便，选取 $\vec{v}$ 为 $P-Q$，以便于我们能用一幅图来讨论点的旋转和向量的旋转。回顾我们在 3.3.1 节中的讨论，我们已经说明向量可相对于另一个向量分解为平行和垂直分量：我们在此将 $\vec{v}$ 投影到 $\hat{u}$，得到 $\vec{v}_\|$ 和 $\vec{v}_\perp$。注意，由于向量与位置无关，我们可将 $\vec{v}$ 画成从旋转轴上的 Q 开始，以使图形更易于理解。

为了有助于理解以下的内容，参见图 4.13，其中显示了垂直于 $\hat{u}$ 且包含 $P=Q+\vec{v}$ 的平面 $\mathcal{A}$。

我们据此可以做如下的推论：

$$\begin{aligned} T(\vec{v}_\perp) &= T(\vec{v})_\perp \\ &= (\cos\theta)\vec{v}_\perp + (\sin\theta)\hat{u} \times \vec{v}_\perp \end{aligned} \tag{4.25}$$

以及

$$
\begin{aligned}
T(\vec{v}_\parallel) &= T(\vec{v})_\parallel \\
&= \vec{v}_\parallel
\end{aligned}
\tag{4.26}
$$

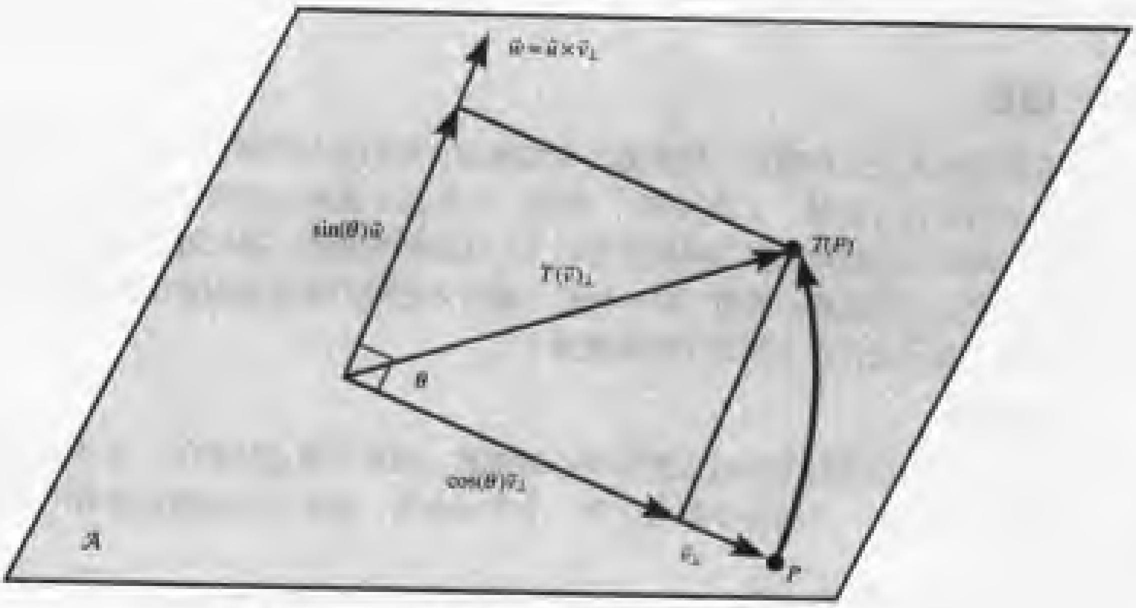


图 4.13　显示在垂直于 $\hat{u}$ 且包含 P 的平面 $\mathcal{A}$ 上的一般旋转

由于 $\vec{v} = \vec{v}_\parallel + \vec{v}_\perp$，并且 T 是线性变换，因此可得

$$T(\vec{v}) = T(\vec{v}_\parallel) + T(\vec{v}_\perp)$$

我们将平行和垂直向量分量的定义代入方程（4.25）和方程（4.26），并将它们扩展，可得

$$
\begin{aligned}
T(\vec{v}) &= (\vec{v}\cdot\hat{u}) + (\cos\theta)(\vec{v} - (\vec{v}\cdot\hat{u})\hat{u}) + (\sin\theta)(\hat{u}\times(\vec{v} - (\vec{v}\cdot\hat{u})\hat{u})) \\
&= (\cos\theta)\vec{v} + (1-\cos\theta)(\vec{v}\cdot\hat{u})\hat{u} + (\sin\theta)(\hat{u}\times\vec{v})
\end{aligned}
$$

该式有时称为旋转公式（rotation formula）或罗德里格斯公式（Rodriguez's formula）（Hecker 1997）。

最后，我们将用该公式和点与向量的加法定义来推论点的旋转公式：

$$
\begin{aligned}
T(P) &= Q + T(P - Q) \\
&= Q + T(\vec{v}) \\
&= Q + (\cos\theta)\vec{v} + (1-\cos\theta)(\vec{v}\cdot\hat{u})\hat{u} + (\sin\theta)(\hat{u}\times\vec{v})
\end{aligned}
$$

如果我们要得到上述公式的矩阵表示，就需要从这些方程中“抽出”向量来。设

$$\mathrm{T}_{\hat{u},\theta} = (\cos\theta)\mathbf{I} + (1-\cos\theta)\vec{u}\otimes\vec{u} + (\sin\theta)\tilde{u}$$

为T的左上角子矩阵（回顾我们在 4.4.2 节中讨论过的内容，即 $\tilde{u}$ 是与 $\hat{u}$ 相关的点积的反对称矩阵）。那么，我们的总变换为

$$\mathrm{T} = \begin{bmatrix} \mathrm{T}_{\hat{u},\theta} & \vec{0}^{\mathrm{T}} \\ Q - Q\mathrm{T}_{\hat{u},\theta} & 1 \end{bmatrix}$$

子矩阵 $\mathrm{T}_{\hat{u},\theta}$ 应该很好理解：它就是向量 $\vec{v}$ 的（线性）变换。但是可能需要稍微解释一下底行。首先，注意底行仅仅影响点的变换，因为向量的矩阵表示的最后一个分量为 0。

显然，向量可被 T 正确变换，因为表示旋转的线性变换分量的$\mathbf{T}_{\vec{u},\theta}$对计算有影响，但是底行没有影响。这等价于假定点 Q 是原点。因此，如果 Q 不是原点，那么我们必须将其平移 Q 与其旋转所得点之间的距离。

4.7.4 缩放

我们分两种情形来处理缩放：均衡的与非均衡的。我们在早些时候（3.3 节）已经说明，有些对点的操作可能是“无意义的”：缩放一个点似乎就是这样的一种操作。正如我们所指出的，缩放只有相对于参考标架才有意义。在这种情形中，我们可以相对于仿射坐标系的原点来定义点的缩放；然而，有一种更一般的方法可用来定义相对于任意原点的缩放和向量缩放（其分量为在不同维中的缩放因子）。

1. 简单缩放

最简单的缩放是以坐标系原点为缩放中心的缩放。缩放可能是均衡的，也可能是非均衡的。在均衡缩放中，一个缩放参数用于每一个基底向量，或者不同的缩放参数用于每一个基底向量。

也同样依次地讨论缩放变换对每一个基底向量的影响（如图 4.14 所示）。

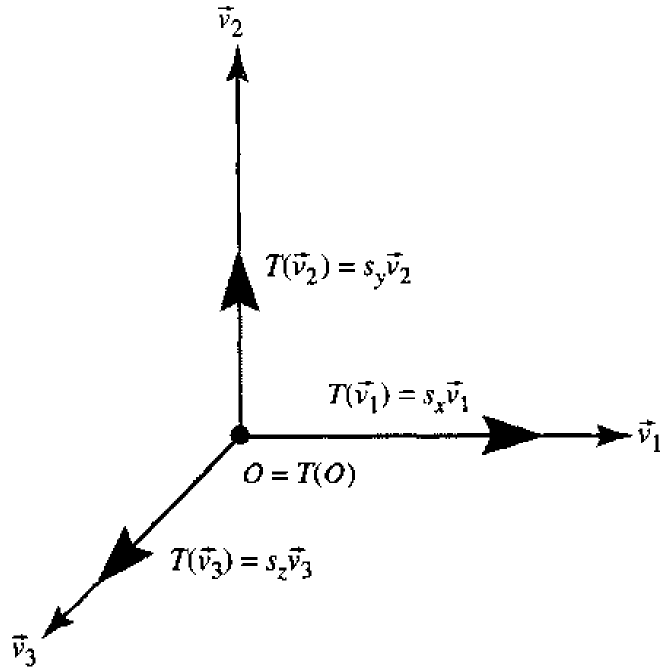


图 4.14　缩放一个坐标系

假定每一个基底向量都分别有一个缩放参数，只是对均衡缩放来说，所有这些参数都具有相同的值。对 $\vec{v}_1$（x 轴）将方程（4.6）应用于 x 轴的矩阵表示：

$$\begin{aligned}T(\vec{v}_1) &= s_x\vec{v}_1\\ &= s_x\,[1 \quad 0 \quad 0 \quad 0]\\ &= [s_x \quad 0 \quad 0 \quad 0]\end{aligned}$$

对 y 轴做相似的操作：

$$\begin{aligned} T(\vec{v}_2) &= s_y\vec{v}_2 \\ &= s_y[0 \quad 1 \quad 0 \quad 0] \\ &= [0 \quad s_y \quad 0 \quad 0] \end{aligned}$$

对 z 轴也做相似的操作：

$$\begin{aligned} T(\vec{v}_3) &= s_z\vec{v}_3 \\ &= s_z[0 \quad 0 \quad 1 \quad 0] \\ &= [0 \quad 0 \quad s_z \quad 0] \end{aligned}$$

由于原点 $\mathcal{O}$ 是缩放的中心，因此它保持不变：

$$\begin{aligned} T(\mathcal{O}) &= \mathcal{O} \\ &= [0 \quad 0 \quad 0 \quad 1] \end{aligned}$$

这样，我们就可以建立一个矩阵，它的各行包含了变换后的基底向量和原点，它实现了相对于原点的简单缩放：

$$T_{s_x,s_y,s_z} = \begin{bmatrix} T(\vec{v}_1) \\ T(\vec{v}_2) \\ T(\vec{v}_3) \\ T(\mathcal{O}) \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

在上述方法中，相对于不是原点的点 Q 的均衡缩放需要如下三个步骤：

第 1 步　平移到原点（即实现平移 $([0 \quad 0 \quad 0] - Q)$）。

第 2 步　应用上述相对原点的旋转。

第 3 步　做与第 1 步相反的平移。

注意，上述操作的次序（两次“额外”的平移）是我们在关于旋转的讨论中尽力想避免的。在此说明这种简单的方法是因为最常用的缩放都是相对于原点的。我们提到实现相对于不是原点的点的缩放所必需的这三个步骤，是为了便于在下一节中讨论更一般的方法。

2. 一般缩放

更一般的缩放方法能实现相对于任意点、沿任意向量指定的方向缩放指定的因子。我们在此分别介绍 Goldman（1987）提出的均衡缩放（uniform scaling）和非均衡缩放（nonuniform scaling）的方法。

（1）均衡缩放

如图 4.15 所示的均衡缩放由缩放原点 Q 和缩放因子 s 所定义。正如 4.3.4 节中所介绍的，向量缩放就是将向量乘以数量，用矩阵来表示，就是用一个缩放因子乘以每一个向量的分量：

$$T(\vec{v}) = s\vec{v} \tag{4.27}$$

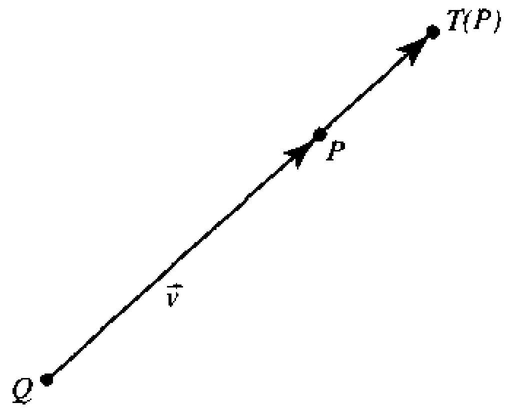


图 4.15　均衡缩放

点的缩放就非常简单了，只需直接应用向量缩放及点与向量的加法和减法的定义就能实现：

$$\begin{aligned} T(P) &= Q + T(\vec{v}) \\ &= Q + s\vec{v} \\ &= Q + s(P - Q) \\ &= sP + (1-s)Q \end{aligned} \tag{4.28}$$

将这些向量代数方程转化为矩阵是很直接的，其方法类似于得到旋转矩阵的方法。如果要旋转一个向量，显然只需考虑方程（4.27），我们需要填充左上角的$n \times n$子矩阵，以缩放每一个分量：

$$\mathrm{T}_s = s\mathbf{I}$$

对于点的缩放，我们需要用方程（4.28）的最右边来填充底行，即矩阵的平移部分。由此得到的结果矩阵为：

$$\mathrm{T}_{s,Q} = \begin{bmatrix} \mathrm{T}_s & \vec{0}^{\mathrm{T}} \\ (1-s)Q & 1 \end{bmatrix}$$

（2）非均衡缩放

非均衡缩放的一般情形稍微复杂一些。与均衡缩放的情况一样，我们有一个缩放原点Q和缩放因子s，但是同时还需要指定一个表示为一个（单位）向量$\hat{u}$的缩放方向，如图 4.16 所示。

为了说明如何缩放向量$\vec{v}$，我们将它投影到$\hat{u}$，分别得到它的垂直分量$\vec{v}_{\perp}$和平行分量和$\vec{v}_{\parallel}$。从图中可明显地看出，

$$T(\vec{v}_{\perp}) = \vec{v}_{\perp}$$

$$T(\vec{v}_{\parallel}) = s\vec{v}_{\parallel}$$

根据向量的加法定义，并将上述方程相减，可得

$$T(\vec{v}) = T(\vec{v}_\perp) + T(\vec{v}_\parallel)$$
$$= \vec{v}_\perp + s\vec{v}_\parallel$$

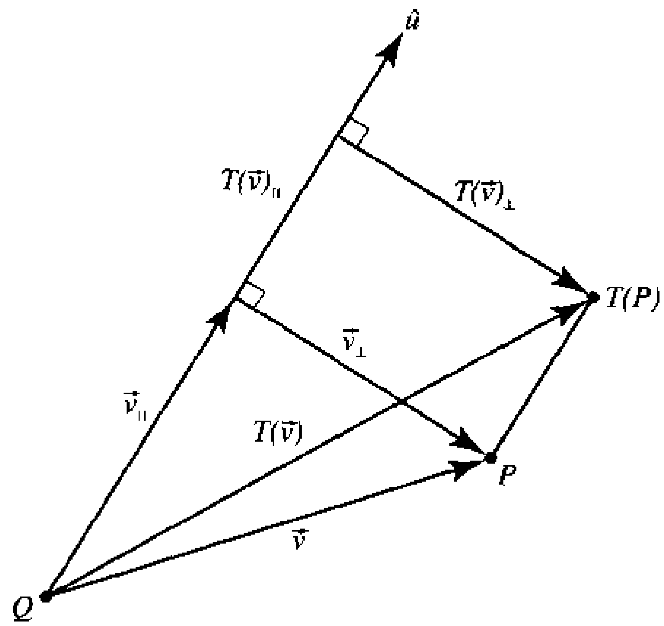


图 4.16　非均衡缩放

如果我们将垂直分量和平行分量的定义（用关于 $\vec{v}$ 和 $\hat{u}$ 的运算）代入，可得

$$\begin{aligned} T(\vec{v}) &= \vec{v}_\perp + s\vec{v}_\parallel \\ &= \vec{v} - (\vec{v}\cdot\hat{u})\hat{u} + s(\vec{v}\cdot\hat{u})\hat{u} \\ &= \vec{v} + (s-1)(\vec{v}\cdot\hat{u})\hat{u} \end{aligned} \tag{4.29}$$

在上式中，我们可以使用点与向量的加法定义，代入上式：

$$\begin{aligned} T(P) &= Q + T(\vec{v}) \\ &= Q + T(P - Q) \\ &= P + (s-1)((P-Q)\cdot\hat{u})\hat{u} \end{aligned} \tag{4.30}$$

我们还是先处理实现了线性变换的左上角的$n \times n$子矩阵，只需从方程（4.29）中减去 $\vec{v}$：

$$\mathrm{T}_{s,\hat{u}} = \mathrm{I} - (1-s)(\hat{u}\otimes\hat{u})$$

对于点的非均衡缩放，我们可从方程（4.30）中抽取P，得到我们需要的矩阵：

$$\mathbf{T}_{s,Q,\hat{u}} = \begin{bmatrix} \mathrm{T}_{s,\hat{u}} & \vec{0}^{\mathrm{T}} \\ (1-s)(Q\cdot\hat{u})\hat{u} & 1 \end{bmatrix} \tag{4.31}$$

术语“非均衡缩放”指的就是s_x, s_y 和 s_z 的值不全相同的“简单缩放”，这里展示的构造并不会使人产生不同的理解。考虑要缩放向量$\hat{u} = [1 \quad 0 \quad 0]$的情形，我们有

$$\mathrm{T}_{s,\hat{u}} = \mathrm{I} - (1-s)(\hat{u}\otimes\hat{u})$$
$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - (1-s)\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} s & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

但是，如果要缩放的方向是 $\vec{u} = [1 \quad 1 \quad 0]$，其规范向量是 $\hat{u} = [\frac{1}{\sqrt{2}} \quad \frac{1}{\sqrt{2}} \quad 0]$，则在这种情形中，我们有

$$\mathbf{T}_{s,\hat{u}} = \mathbf{I} - (1-s)(\hat{u} \otimes \hat{u})$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - (1-s) \begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 + \frac{-1+s}{2} & \frac{-1+s}{2} & 0 \\ \frac{-1+s}{2} & 1 + \frac{-1+s}{2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

上式清楚地说明，非均衡缩放确实比“简单缩放”更一般化。

4.7.5 反射

反射是一种相对于一条直线（在二维空间中）或一个平面（在三维空间中）的镜像点的变换。图 4.17 显示了二维空间的情形。反射的一个特别重要的性质是反转方向，从图中可以看出这一点。

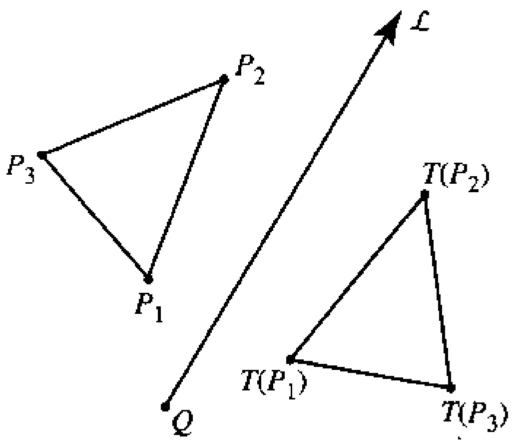


图 4.17　镜像

1. 简单反射

最简单的反射是相对于一条通过原点且在一个基底向量的方向上的直线（在二维空间中）或者一个通过原点且由三个基底向量中的两个所定义的平面（即 xy 平面，xz 平面或 yz 平面）的反射。为了说明得更清晰，并且描述如何将反射扩展到三维，我们先说明二维反射。

设反射是相对 y 轴的。我们仍然依次考虑变换对各基底向量和原点的影响，并且通过将变换所得的向量和原点作为行来建立矩阵。

相对 y 轴的反射并不影响基底向量 $\vec{v}_2$，因此可得

$$T(\vec{v}_2) = \vec{v}_2$$

$$= [0 \quad 1 \quad 0]$$

然而，变换T可影响基底向量$\vec{v}_1$；操作只是简单地反转其方向，因此我们有

$$\begin{aligned} T(\vec{v}_1) &= -\vec{v}_1 \\ &= [-1 \quad 0 \quad 0] \end{aligned}$$

最终，变换T并不影响原点$\mathcal{O}$，因此可得

$$\begin{aligned} T(\mathcal{O}) &= \mathcal{O} \\ &= [0 \quad 0 \quad 1] \end{aligned}$$

因此我们的变换矩阵为

$$\mathbf{T} = \begin{bmatrix} T(\vec{v}_1) \\ T(\vec{v}_2) \\ T(\mathcal{O}) \end{bmatrix} = \begin{bmatrix} -\vec{v}_1 \\ \vec{v}_2 \\ \mathcal{O} \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

如图 4.18 所示。

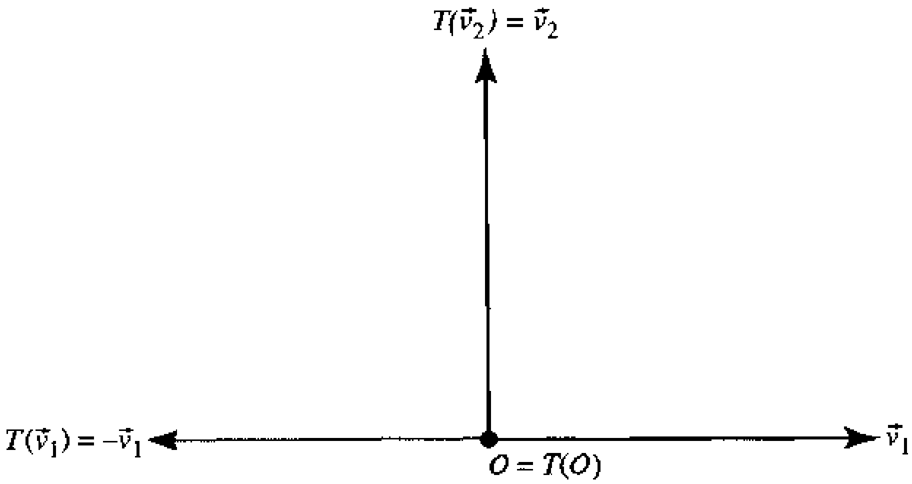


图 4.18　二维空间中的简单反射

可以很简单地扩展到三维空间中的简单反射。不再简单地相对于一个基底向量反射，我们现在需要相对一对基底向量反射，这对基底向量定义一个通过原点的平面（即xy平面，xz平面或yz平面中的一个）。在相对于$\vec{v}_2$（y轴）的二维反射的例子中，我们看到反射变换不影响基底向量$\vec{v}_2$，但是反转$\vec{v}_1$；在三维空间中，相对于xz平面（如图 4.19 所示）的反射将不影响$\vec{v}_1$或$\vec{v}_3$，但是会反转$\vec{v}_2$（y—轴），我们可得到如下的变换矩阵

$$\mathbf{T} = \begin{bmatrix} T(\vec{v}_1) \\ T(\vec{v}_2) \\ T(\vec{v}_3) \\ T(\mathcal{O}) \end{bmatrix}$$

$$
= \begin{bmatrix} \vec{v}_1 \\ -\vec{v}_2 \\ \vec{v}_3 \\ \mathcal{O} \end{bmatrix}
$$

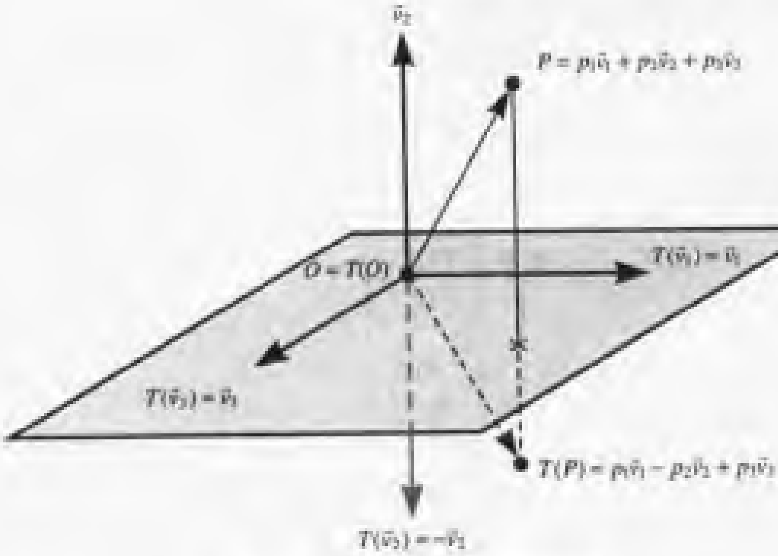


图 4.19　三维空间中的简单反射

2．一般反射

一般反射可定义为相对任意直线（在二维空间中）或任意平面（在三维空间中）的反射。为了方便说明，如图 4.17 所示，我们用直线上的一个点 Q 和一个向量来定义反射直线。如图 4.20 所示，用其上的一个点 Q 和一个法线向量 $\hat{n}$ 来定义三维空间的反射平面。

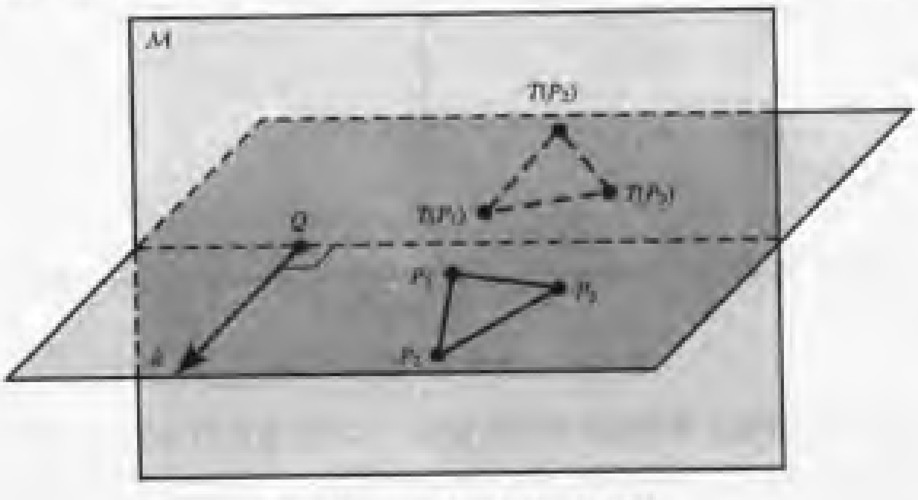


图 4.20　三维空间中的一般反射

二维的情形如图 4.21 所示。我们相对由原点 Q 和一个方向向量 $\hat{d}$ 所定义的任意方向的直线进行反射。

与前面一样，首先来看看向量的反射。如果将 $\vec{v}$ 投影到 $\hat{d}^{\perp}$，可分别得到垂直分量 $\vec{v}_{\perp}$ 和平行分量和 $\vec{v}_{\parallel}$。通过观察可知 $\vec{v}_{\perp}$ 是平行于 $\hat{d}$ 的，$\vec{v}_{\parallel}$ 是在 $\hat{d}^{\perp}$ 上的，由定义可知 $\hat{d}^{\perp}$ 是垂直于 $\hat{d}$ 的，因此，我们可以简单地得出如下的结论

$$T(\vec{v}_\perp) = \vec{v}_\perp$$

$$T(\vec{v}_\parallel) = -\vec{v}_\parallel$$

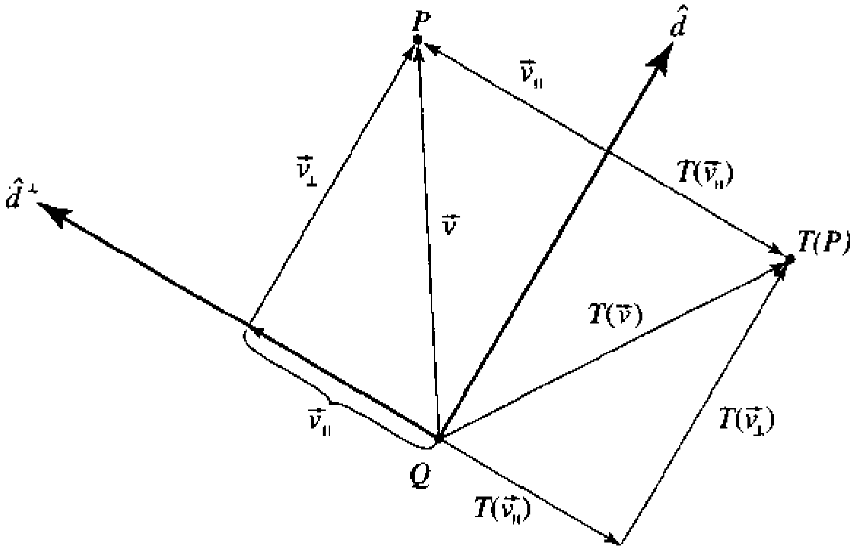


图 4.21　二维空间中的镜像

根据向量加法的定义，将上述两个方程相减，并应用向量投影的定义，我们可以得到如下的结论

$$\begin{aligned} T(\vec{v}) &= T(\vec{v}_\perp) + T(\vec{v}_\parallel) \\ &= \vec{v}_\perp - \vec{v}_\parallel \\ &= \vec{v} - 2\vec{v}_\parallel \\ &= \vec{v} - 2(\vec{v} \cdot \hat{d}^\perp)\hat{d}^\perp \end{aligned} \tag{4.32}$$

与前面的方法一样，我们可以利用点与向量的加法的定义，将点做如下的变换：

$$\begin{aligned} T(P) &= Q + T(\vec{v}) \\ &= Q + T(P - Q) \\ &= P - 2((P - Q) \cdot \hat{d}^\perp)\hat{u} \end{aligned}$$

我们还是先通过简单地从方程（4.32）中抽取 $\vec{v}$，来处理实现了线性变换的左上角的 $n \times n$ 矩阵：

$$\mathrm{T}_{\hat{d}} = [\mathrm{I} - 2(\hat{d}^\perp \otimes \hat{d}^\perp)]$$

对于矩阵的平移部分可以像以前一样地求得，从而得到一个完整的反射矩阵：

$$\mathrm{T}_{\hat{d},Q} = \begin{bmatrix} \mathrm{T}_{\hat{d}} & \vec{0}^{\mathrm{T}} \\ 2(Q \cdot \hat{d}^\perp)\hat{d}^\perp & 1 \end{bmatrix}$$

你可能觉得奇怪，我们为什么选择将 $\vec{v}$ 投影到 $\hat{d}^\perp$ 上，而不选择投影到 $\hat{d}$ 上。原因是，如果我们的反射所相对的平面是由其上的点 Q 和一条法线 $\hat{n}$ 所表示的，那么我们就可以直接将其扩展到三维空间，如图 4.22 所示。

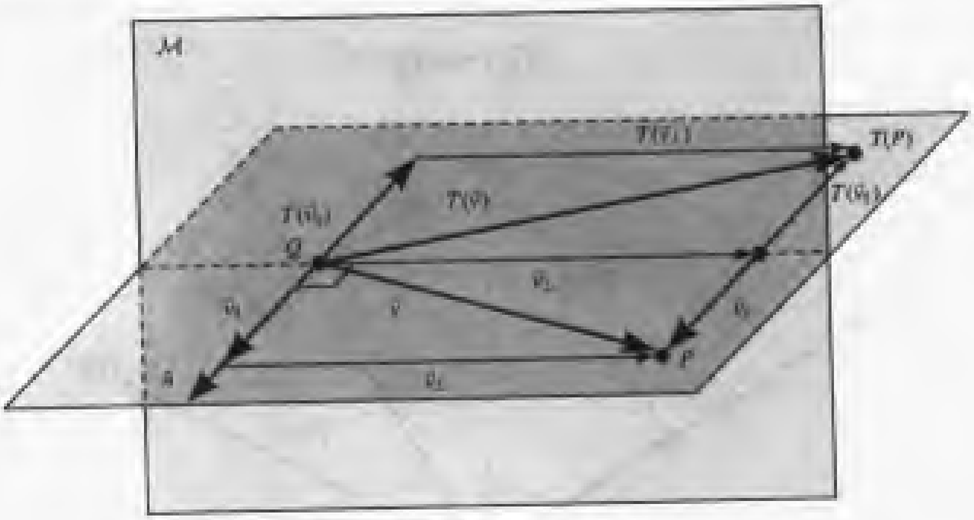


图 4.22　三维空间中的镜像

用与二维空间相同的建立方法，可得到如下的三维空间的反射矩阵

$$\mathbf{T}_{\hat{n},Q}=\begin{bmatrix}\mathbf{T}_{\hat{n}} & \vec{0}^{\mathrm{T}} \\ 2(Q\cdot\hat{n})\hat{n} & 1\end{bmatrix}$$

4.7.6　剪切

剪切变换是一种更加有趣的仿射变换。图 4.23 显示了两种不同的二维剪切的例子，即相对于每一个基底向量及坐标轴的剪切。剪切没有其他反射变换常用，但在排版印刷中经常见到的一个应用实例，即斜体字，就是图 4.23 的右图所示的一种剪切（虽然可能其剪切角度要小一些）。

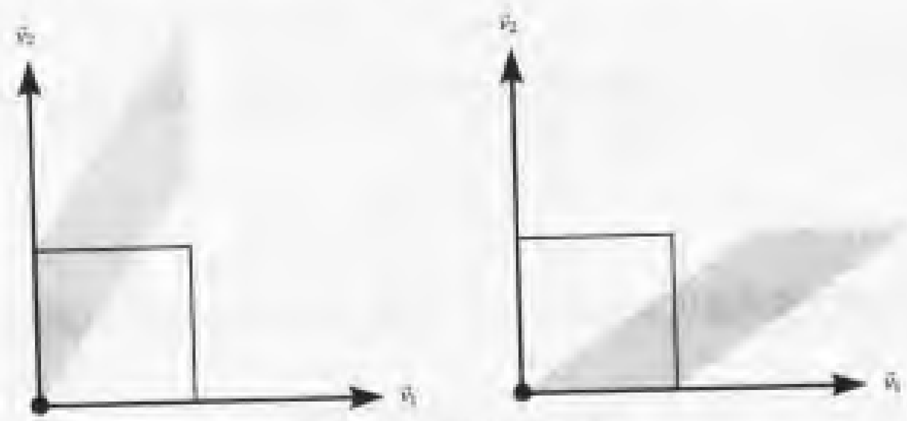


图 4.23　二维空间中的剪切

1. 简单剪切

一般地，剪切可以沿任意线（在二维空间中）或正交于任意平面（在三维空间中）进行，但正如讨论其他变换的方式一样，我们首先讨论简单的情形。简单的剪切沿基底向量并通过原点来进行。如图 4.23 所示，剪切将矩形变换成平行四边形；然而，平行四边形的面积必须与原来的矩形的面积相同。任何的平行四边形的面积为*底*×*高*，这清楚地说明了

为什么简单剪切保持沿轴的矩阵的底或高。

有多种方法可以用来说明简单剪切。我们在此采用的方法是指定进行剪切的轴和剪切角度。其他的书籍，如 Möller 和 Haines（1999），一般指定剪切比例而不是剪切角度。

我们还是通过依次考虑变换对坐标系的基底向量和原点的影响来建立变换矩阵。下面将说明一个沿 x 轴的剪切（图 4.23 的右图），表示为 $T_{xy,\theta}$（当我们说明三维剪切时，我们会把将下标说明为 xy 而不仅仅是 x 的理由解释得更清楚一些）。

首先，我们考虑 $\vec{v}_1$ 为 x 轴时的剪切 $T_{xy,\theta}$。从图 4.24 中可以看出，x 轴没有变换：

$$\begin{aligned} T_{xy,\theta}(\vec{v}_1) &= \vec{v}_1 \\ &= [1 \quad 0 \quad 0] \end{aligned}$$

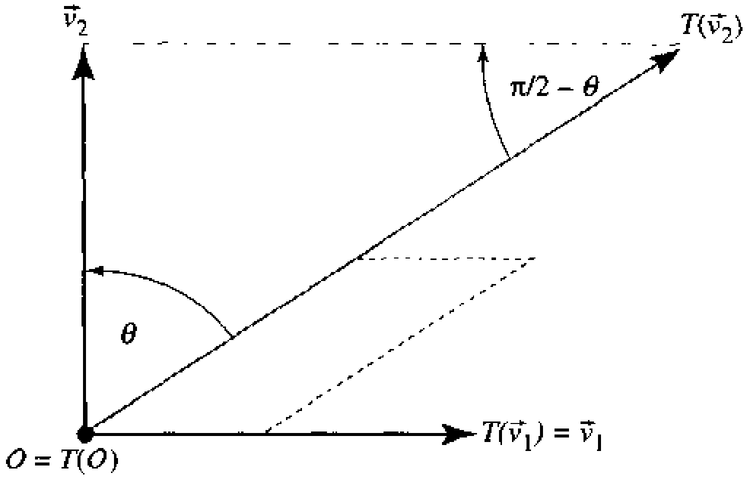


图 4.24　$T_{xy,\theta}$

$\vec{v}_2$ 为 y 轴时的剪切 $T_{xy,\theta}$ 可以通过如下的方法来计算：如果我们考虑原点 $\mathcal{O}$ 和点 $\mathcal{O}+\vec{v}_2$ 以及 $\mathcal{O}+T(\vec{v}_2)$ 所构成的直角三角形，可以设顶点为 $\mathcal{O}$ 的角为 θ。由于我们假设使用标准的欧几里得基底，因此 $\|\vec{v}_2\|=1$。我们可以简单地应用三角几何来推导 $\vec{v}_2$ 的剪切 $T_{xy,\theta}$：

$$T_{xy,\theta}(\vec{v}_2) = [\tan\theta \quad 1 \quad 0]$$

最后，由于变换不影响原点，因此，

$$\begin{aligned} T(\mathcal{O}) &= \mathcal{O} \\ &= [0 \quad 0 \quad 1] \end{aligned}$$

所以变换矩阵为

$$\begin{aligned} \mathbf{T}_{xy,\theta} &= \begin{bmatrix} T(\vec{v}_1) \\ T(\vec{v}_2) \\ T(\mathcal{O}) \end{bmatrix} \\ &= \begin{bmatrix} 1 & 0 & 0 \\ \tan\theta & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

相似的剪切 $T_{xy,\theta}$（在 y 方向上的剪切）为

$$\mathrm{T}_{yx,\theta}=\begin{bmatrix}1 & \tan\theta & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

在三维空间中，存在6种剪切。如果我们设

$$H_{\eta\gamma}=\tan\theta_{\eta\gamma}$$

其中η是指剪切所改变的坐标，γ是指进行剪切的坐标。那么，可以建立如下的方法，以说明获取需要的剪切方向的剪切因子定位：

$$\begin{bmatrix}1 & H_{yx} & H_{zx} & 0\\ H_{xy} & 1 & H_{zy} & 0\\ H_{xz} & H_{yz} & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}$$

这些H因子中只有一个应该为非零。两个或多个剪切的组合必须通过求得每一个剪切矩阵并将这些矩阵相乘在一起来实现。

2. 一般剪切

一种更一般的剪切可以由剪切平面S、S中的单位向量$\hat{v}$和剪切角θ来定义。平面S由其上的点Q和法线$\hat{n}$所定义。剪切变换移动空间中的一组平行的平面并使它们保持平行；其中有一个平面（定义剪切的平面）不会移动。Goldman（1991）给出了一种剪切变换矩阵的建立方法（参见图4.25）：对任何点P，设P'为其在S上的正交投影，并建立点$P''=P+t\hat{v}$，使得$\angle P''P'P=\theta$。如果我们将剪切应用于P，可以得到P''。变换为

$$T_{Q,\hat{n},\hat{v},\theta}=\begin{bmatrix}\mathrm{I}+\tan\theta(\hat{n}\otimes\hat{v}) & \vec{0}^{\mathrm{T}}\\ -(Q\cdot\hat{n})\hat{v} & 1\end{bmatrix}$$

Goldman注意到该矩阵的行列式为1，因此这种剪切变换保持体积不变（注意$-\pi/2<\theta<\pi/2$）。

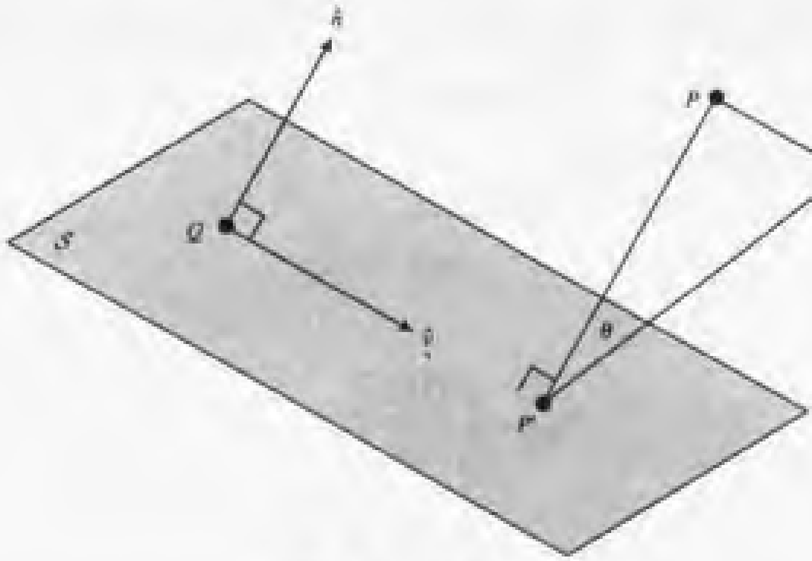


图4.25 一般剪切说明

一般的三维剪切矩阵可沿不同的线来建立，这样可能更简单，向量$\hat{v}$和$\hat{n}$，以及点Q定

义了一个仿射坐标秒(第三个基底向量为 $\hat{v}\times\hat{n}$)。如果我们有点 $P=Q+y_1\hat{n}+y_2\hat{v}+y_3(\hat{n}\times\hat{v})$，那么，剪切操作可以将该点映射为

$$\begin{aligned}T(P)&=T(Q)+y_1T(\hat{n})+y_2T(\hat{v})+y_3T(\hat{n}\times\hat{v})\\&=Q+y_1(\hat{n}+\tan\theta\hat{v})+y_2\hat{v}+y_3(\hat{n}\times\hat{v})\\&=Q+y_1\hat{n}+(y_2+y_1\tan\theta)\hat{v}+y_3(\hat{n}\times\hat{v})\end{aligned}$$

与仿射坐标系相关的矩阵为

$$\begin{bmatrix}1 & \tan\theta & 0 & 0\\0 & 1 & 0 & 0\\0 & 0 & 1 & 0\\Q_x & Q_y & Q_z & 1\end{bmatrix}$$

读者应该可以由上节的内容中认出其左上方的 3×3 方块矩阵 **H** 为二维空间的 y 剪切矩阵 $T_{xy,\theta}$（基底向量 $\hat{v}$ 相当于该坐标系的 y 轴）。

利用标准的欧几里得基底，如果我们有旋转矩阵

$$\mathbf{R}=\begin{bmatrix}\hat{n}\\\hat{v}\\\hat{n}\times\hat{v}\end{bmatrix}$$

以及

$$\vec{y}=[\,y_1\quad y_2\quad y_3\,]$$

那么

$$P=Q+\vec{y}\mathbf{R}$$

且有

$$\begin{aligned}T(P)&=Q+\vec{y}\mathbf{HR}\\&=Q+(P-Q)\mathbf{R}^{\mathrm{T}}\mathbf{HR}\end{aligned}$$

矩阵 **T** 的变换矩阵为

$$\mathbf{T}_{\hat{n},\hat{v},Q,\theta}=\begin{bmatrix}\mathbf{R}^{\mathrm{T}}\mathbf{HR} & \vec{0}^{\mathrm{T}}\\Q(\mathrm{I}_3-\mathbf{R}^{\mathrm{T}}\mathbf{HR}) & 1\end{bmatrix}$$

4.8 投影

投影是一类具有如下性质的变换，即将点从一个 n 维坐标系变换到另一个维数小于 n（一般为 $n-1$）的坐标系中。投影变换在计算机图形学中最重要的应用是用于图形显示系统和工具库的管道着色，三维物体在着色和显示在终端屏幕上之前先被投影到一个平面中。

投影类变换包括两个在计算机图形学中非常有用的子类：平行投影和透视投影。投影可定义为连接物体上每一点的线与一个平面的交点。在平行投影中，所有这些投影线都是平行的，而在透视投影中，这些投影线相汇于同一点，该点被称为投影中心。正如 Foley 等（1996，6.1 节）所指出的，可以将平行投影的投影中心看成是无穷远处的一个点。

在下面的几节中，我们将介绍如何只用向量代数技术来建立正射投影和透视投影的变换矩阵。对于平行投影的不同子类的完整的处理方法，以及用于创建视角变换的平行投影和透视投影的变换矩阵的建立，请参阅 Foley 等（1996，第 6 章）。

4.8.1 正射投影

正射投影（又叫做正交投影）是我们将讨论的投影中最简单的类型：它仅仅包含将一个垂直面上的点和线投影到一个平面上，如图 4.26 所示。与我们在讨论镜像变换时一样，我们用其上的一个点和其（单位）法线向量来定义一个平面。

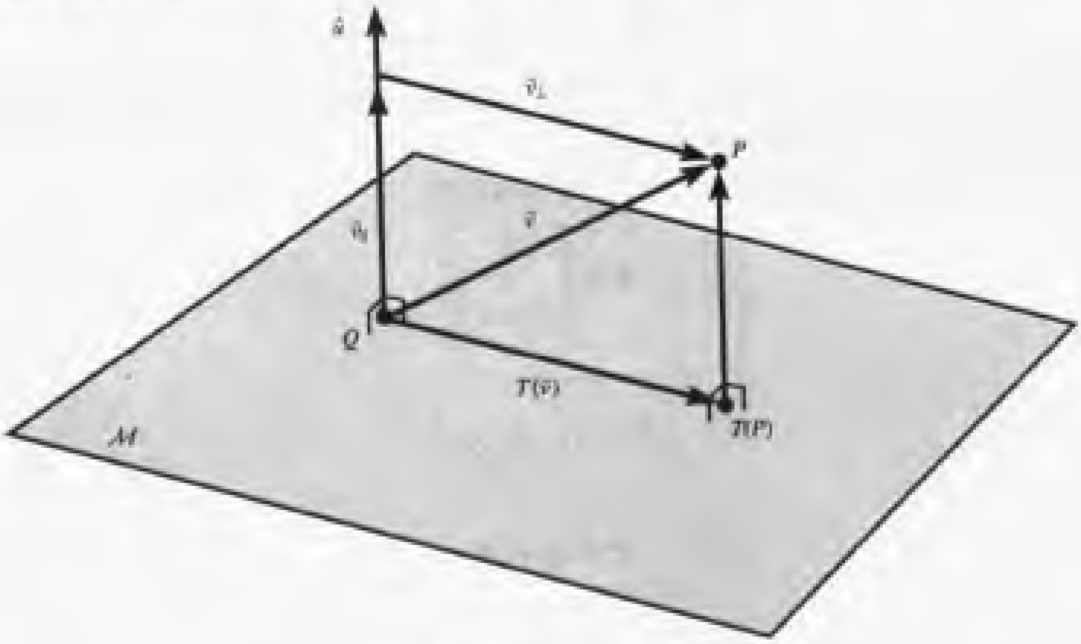


图 4.26　正射（正交）投影

向量的正射投影就是我们一直使用的那种投影：将向量 $\vec{v}$ 投影到法线 $\hat{n}$ 上，以得到垂直分量和平行分量，注意，由于 $\vec{v}_{\perp}$ 是与位置无关的，点 Q 的相对位置是不需理会的，因此，可得

$$
\begin{aligned}
T(\vec{v}) &= \vec{v}_{\perp} \\
&= \vec{v} - (\vec{v}\cdot\hat{u})\hat{u}
\end{aligned}
\tag{4.33}
$$

类似地，可简单地得到点 P 的变换：

$$
\begin{aligned}
T(P) &= Q + T(\vec{v}) \\
&= Q + T(P - Q) \\
&= P - ((P - Q)\cdot\hat{u})\hat{u}
\end{aligned}
\tag{4.34}
$$

上式的矩阵表示可通过如下的方法来得到，即从方程（4.33）中析出因子 $\vec{v}$ 并得到左上角的 $n \times n$ 子矩阵：

$$\mathrm{T}_{\hat{u}} = \mathrm{I} - (\hat{u} \otimes \hat{u})$$

最底行的计算与从前的方法相同，即从上式中提出公因子，并从方程（4.34）中提出点 P：

$$T_{\hat{u},Q} = \begin{bmatrix} T_{\hat{u}} & \vec{0}^T \\ (Q \cdot \hat{u})\hat{u} & 1 \end{bmatrix}$$

4.8.2　斜轴投影

斜轴（或平行）投影就是正射投影的一般化——投影是平行的，但是平面并不垂直于投影"射线"。同样，用其上的一个点 Q 和其（单位）法线向量 $\hat{u}$ 来定义投影平面，然而，由于法线 $\hat{u}$ 不再定义投影方向，因此需要指定另一个（单位）向量 $\hat{w}$ 作为投影方向，如图 4.27 所示。

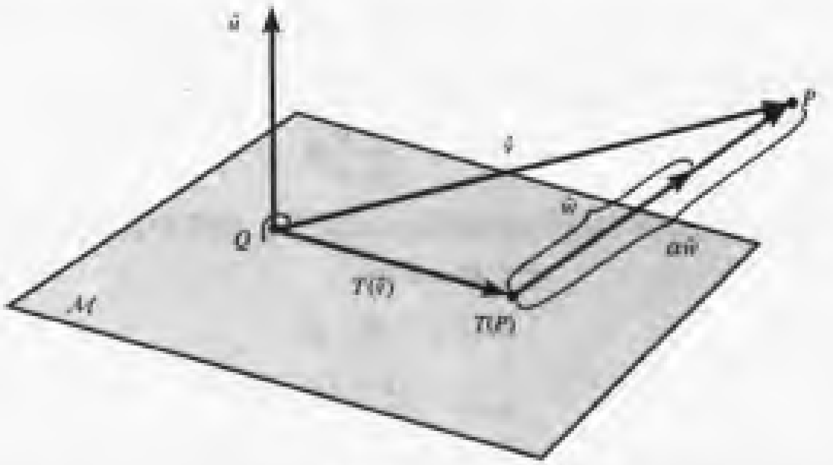


图 4.27　斜轴投影

该图的切面图（如图 4.28 所示）将帮助我们解释如何确定向量的变换。从观察下式开始：

$$\vec{v} = T(\vec{v}) + \alpha\hat{w}$$

可以将其改写为

$$T(\vec{v}) = \vec{v} - \alpha\hat{w} \tag{4.35}$$

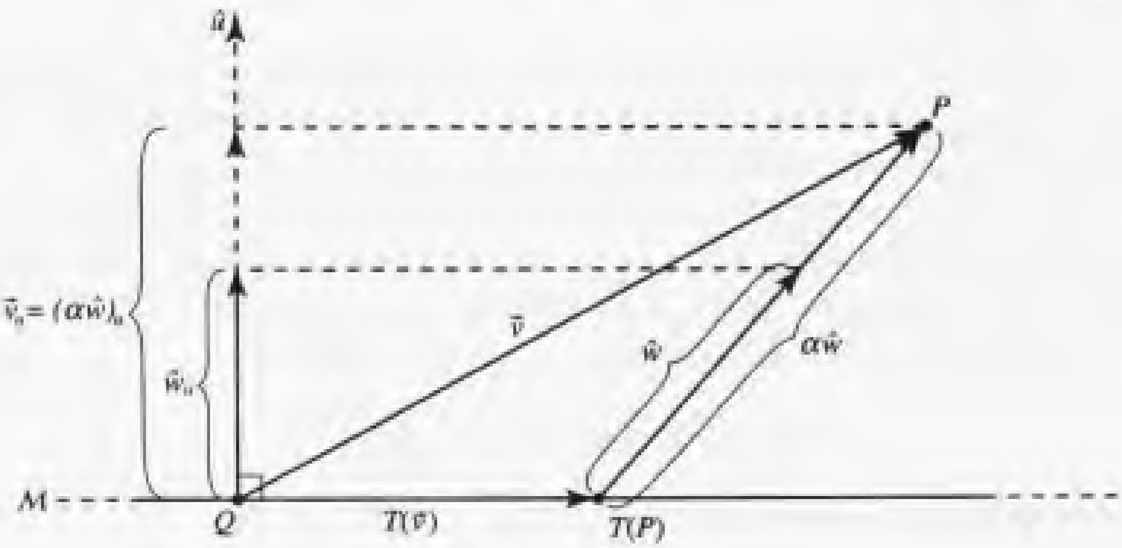


图 4.28　斜轴投影的切面

显然，需要计算的是α。如果注意到下述等式，其计算是相当直接的：

$$\|\vec{v}_{\|}\| = \|(\alpha\hat{w})_{\|}\|$$

可以将其改写为

$$\frac{\|\vec{v}_{\|}\|}{\|\hat{w}_{\|}\|} = \frac{\|(\alpha\hat{w})_{\|}\|}{\|\hat{w}_{\|}\|} = \alpha$$

可用点积的定义将其改写为

$$\alpha = \frac{\vec{v} \cdot \hat{u}}{\hat{w} \cdot \hat{u}} \tag{4.36}$$

然后，可将方程（4.36）代入方程（4.35），并得到向量变换：

$$T(\vec{v}) = \vec{v} - \frac{\vec{v} \cdot \hat{u}}{\hat{w} \cdot \hat{u}}\hat{w} \tag{4.37}$$

再像过去一样地应用点与向量的加法和减法的定义，以得到点变换的公式：

$$\begin{aligned} P &= Q + T(\vec{v}) \\ &= Q + T(P - Q) \\ &= P - \frac{((P - Q) \cdot \hat{u})\hat{w}}{\hat{w} \cdot \hat{u}} \end{aligned} \tag{4.38}$$

为了将其转换为矩阵表示，我们应用常用的技术，即从方程（4.37）中抽出$\vec{v}$，以计算左上角的$n \times n$子矩阵：

$$\mathbf{T}_{\hat{u},\hat{w}} = \mathbf{I} - \frac{(\hat{u} \otimes \hat{w})}{\hat{w} \cdot \hat{u}}$$

为了计算变换矩阵的最底行，从方程（4.38）中抽出$\vec{v}$和点P，完整的矩阵如下所示：

$$\mathbf{T}_{\hat{u},Q,\hat{w}} = \begin{bmatrix} \mathbf{T}_{\hat{u},\hat{w}} & \vec{0}^{\mathrm{T}} \\ \frac{Q \cdot \hat{u}}{\hat{w} \cdot \hat{u}}\hat{w} & 1 \end{bmatrix}$$

4.8.3 透视投影

透视投影不是一种仿射变换，例如它不将平行线映射为平行线。与平行投影和正射投影不同，透视投影并非对所有点和向量都是均衡的，而是有一个投影点或透视点，而且，投影线由物体上的每一个点与其透视点所决定的向量所定义，如图 4.29 所示。

然而，透视投影确实保持了一种称为交比（cross-ratios）的性质。我们知道，反射映射保持比例：给定三个共线的点P，R和Q，P与R之间的距离（记为$\overline{PR}$）和R与Q之间的距离（$\overline{RQ}$）之比等于$\overline{T(P)T(R)}$与$\overline{T(R)T(Q)}$之比（如图 3.36 所示）。

交比的定义如下：给定 4 个共线的点$P, R_1, R_2,$和Q，其交比为

$$\mathrm{CrossRatio}(P, R_1, R_2, Q) = \frac{\overline{PR_1}/\overline{R_1Q}}{\overline{PR_2}/\overline{R_2Q}}$$

如图 4.30 的左图所示。保持交比意味着下式成立：

$$\mathrm{CrossRatio}(P, R_1, R_2, Q) = \mathrm{CrossRatio}(T(P), T(R_1), T(R_2), T(Q))$$

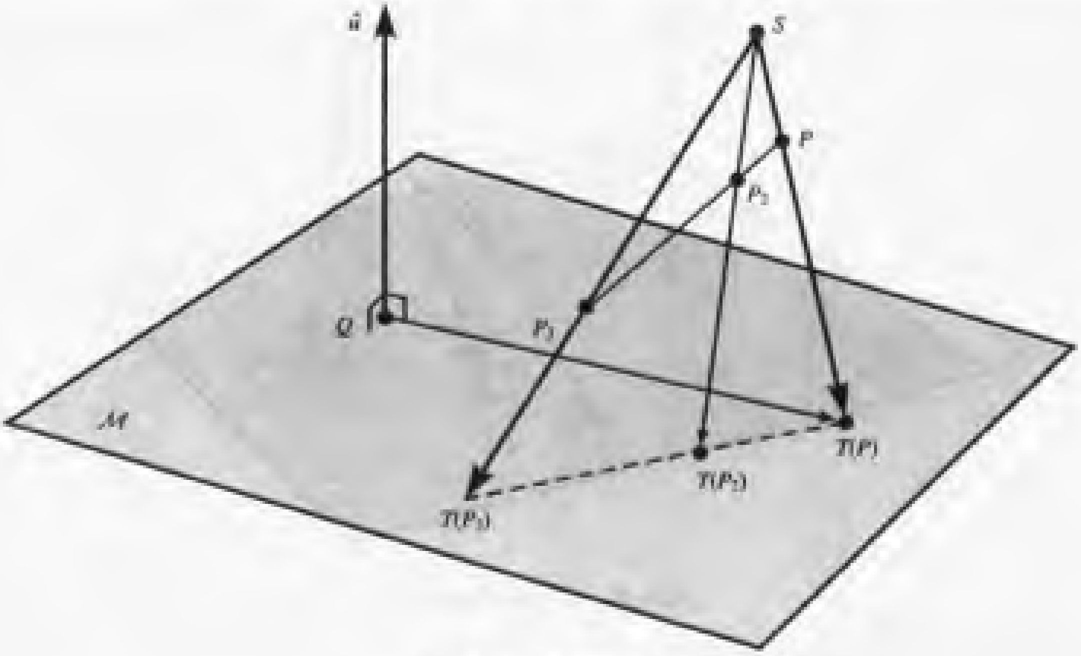


图 4.29　透视投影

如图 4.30 所示。关于交比的更完整介绍可以从如下文献中找到：Farin（1990）和 DeRose（1992）。

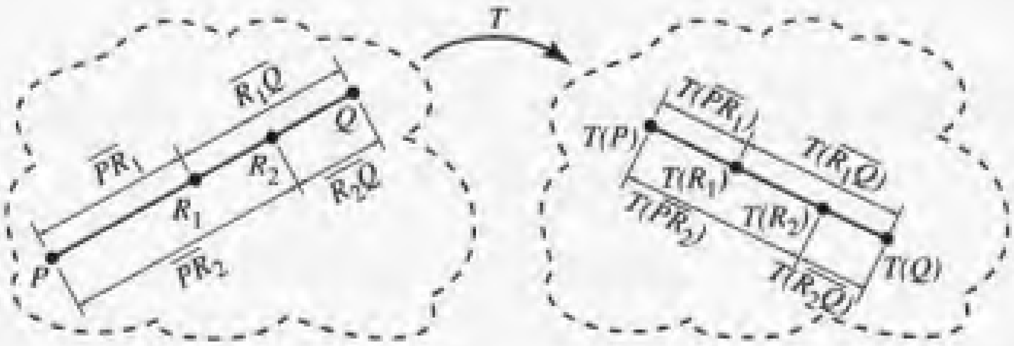


图 4.30　交比

向量的透视投影是不清晰的，理解这一点很重要。如果我们有任意两组点，它们之差确定一个向量，每一组点的平行投影点所决定的向量与原来的向量相同。而在透视投影中，这一点不再成立：如图 4.31 所示，我们有两对这样的点(P_1, P_2)和(P_3, P_4)，每一对点所定义的向量相等，即 $\vec{v}_1 = \vec{v}_2$。然而，如果观察它们的投影点之间的向量（分别为$(T(\vec{v}_1) = T(P_2) - T(P_1)$和$T(\vec{v}_2) = T(P_4) - T(P_3))$，很明显，它们是不相等的。

但是，对于点的透视投影却定义得很清晰。显然，$T(P)$ 是P和S 所确定的向量的投影向量的终点：

$$T(P) = P + \alpha(S - P) \tag{4.39}$$

另一种观察方式如下：

$$T(P) - Q = (P - Q) + \alpha(S - P)$$

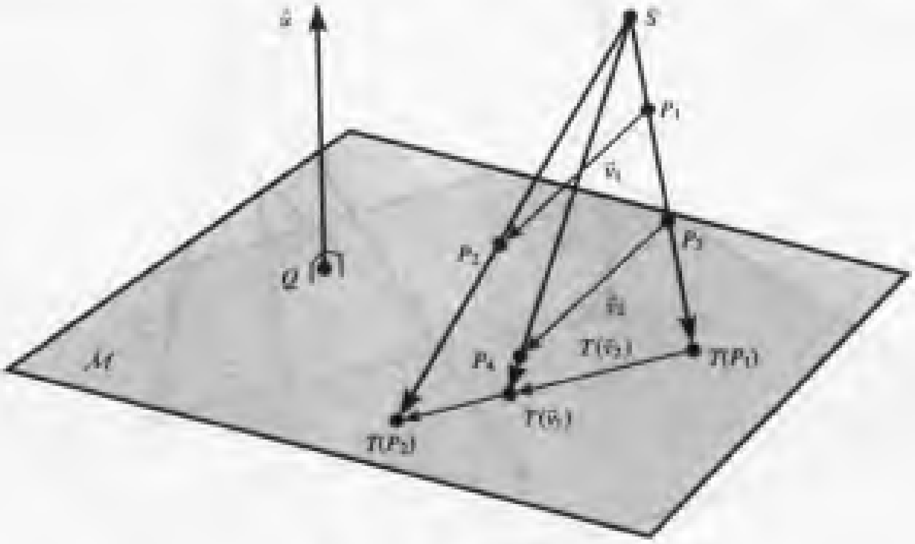


图 4.31　向量的透视映射

该向量也与平面法线垂直，因此，有

$$\begin{aligned}0 &= \hat{u}\cdot((P-Q)+\alpha(S-P)) \\ &= \hat{u}\cdot(P-Q)+\alpha\hat{u}\cdot(S-P)\end{aligned}$$

如果对上式求解α，可得

$$\alpha=\frac{\hat{u}\cdot(Q-P)}{\hat{u}\cdot(S-P)} \tag{4.40}$$

现在可将方程（4.40）代入方程（4.39），并做一点向量运算，可得如下的最终公式：

$$\begin{aligned}T(P) &= P+\frac{(Q-P)\cdot\hat{u}}{(S-P)\cdot\hat{u}}(S-P) \\ &= \frac{((S-Q)\cdot\hat{u})P+((Q-P)\cdot\hat{u})S}{(S-P)\cdot\hat{u}}\end{aligned}$$

其变换矩阵为

$$\mathbf{T}_{\hat{u},Q,S}=\begin{bmatrix}((S-Q)\cdot\hat{u})\mathbf{I}-\hat{u}\otimes S & -\hat{u}^{\mathrm{T}} \\ (Q\cdot\hat{u})S & S\cdot\hat{u}\end{bmatrix}$$

4.9　变换法线向量

我们已经知道，向量可以像点一样通过矩阵乘法来进行变换。该原理似乎可用来处理曲面的法线向量，然而，这样做（一般）是不正确的。由于这并不是很直观，因此在介绍正确的方法之前，先稍微详细一点地解释为什么这样做是错误的。

这里的主要问题是由于非均衡缩放而产生的。Eric Haines（1987）给出的一个例子非常清楚地说明了这一点：考虑三维空间中的一个平面，如果我们从z轴往下看，它将表现为一条角度为 45°的线，如图 4.32 所示。

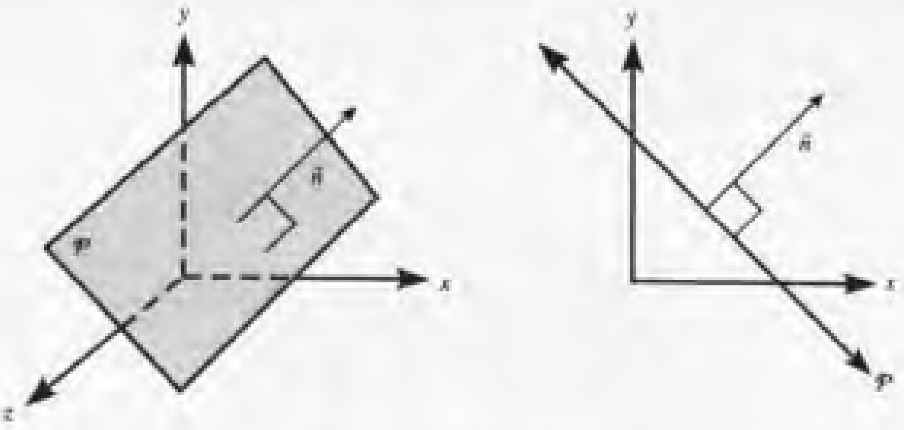


图 4.32　平面 $x + y = k$

该平面的法线为 $\vec{n} = [1 \quad 1 \quad 0]$。假设我们有一个非均衡缩放变换 T，只有 x 进行因子为 2 的缩放。其变换矩阵为

$$\mathrm{T} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

如果对平面和法线进行这种变换，将得到预期的缩放平面。但是，考虑对法线的变换

$$T(\vec{n}) = \vec{n}\mathrm{T} = [1 \quad 1 \quad 0]\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [2 \quad 1 \quad 0]$$

如图 4.33 所示。很清楚，这是不正确的。问题在哪里呢？在于法线并不是一般概念上的真正的向量。实际上，曲面的法线是两个与曲面相切的（线性无关的）向量的叉积：

$$\vec{n} = \vec{u} \times \vec{v}$$

如图 4.34 所示。如果 T 是包含曲面的空间中的线性变换，那么，变换所得的切线为 $T(\vec{u})$

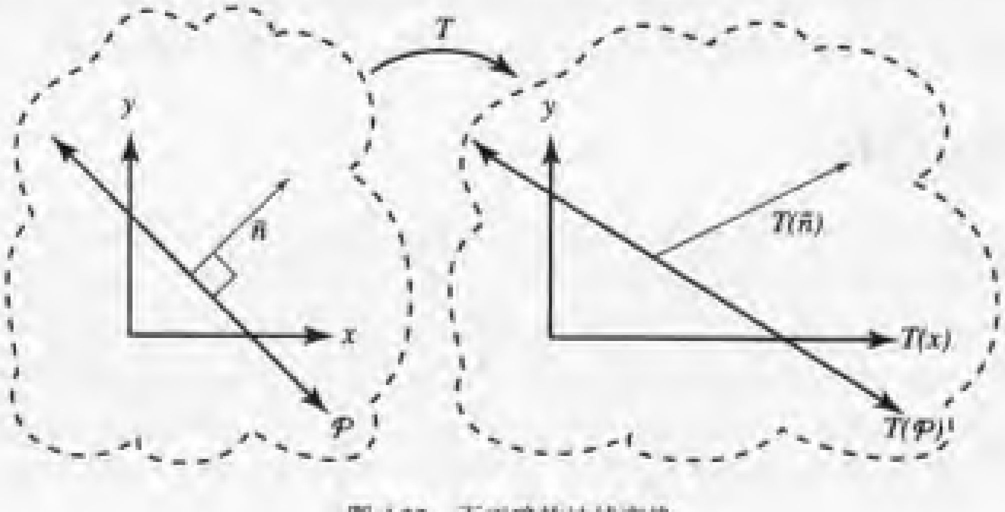


图 4.33　不正确的法线变换

和$T(\vec{v})$，它们都与变换后的曲面相切。变换后的曲面的法线为 $\vec{m} = T(\vec{u}) \times T(\vec{v})$。然而，正如我们见到的，$\vec{m}$ 并不一定就是原来法线变换后的结果$T(\vec{n})$，$T(\vec{u} \times \vec{v}) \neq T(\vec{u}) \times T(\vec{v})$，更一般地，$T(\vec{u} \times \vec{v})$并不一定平行于$T(\vec{u}) \times T(\vec{v})$。

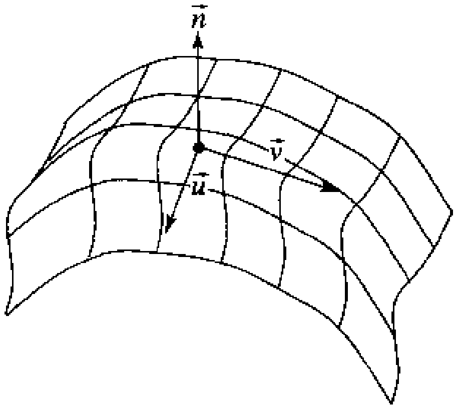


图 4.34　法线表现为曲面切线的叉乘

任何切线向量$\vec{u}$的垂直向量$\vec{n}$都可以表示为

$$\vec{u} \cdot \vec{n} = \vec{u}\vec{n}^{\mathrm{T}} = 0$$

我们要变换后的法线垂直于变换后的切线：

$$T(\vec{u}) \cdot \vec{m} = 0$$

如果对此进行数学运算，可得

$$\begin{aligned} 0 &= \vec{u}\vec{n}^{\mathrm{T}} \\ &= \vec{u}\mathbf{T}\mathbf{T}^{-1}\vec{n}^{\mathrm{T}} \\ &= (\vec{u}\mathbf{T})(\vec{n}(\mathbf{T}^{-1})^{\mathrm{T}})^{\mathrm{T}} \\ &= T(\vec{u}) \cdot (\vec{n}(\mathbf{T}^{-1})^{\mathrm{T}}) \end{aligned}$$

因此，变换后曲面的法线向量为$\vec{m} = \vec{n}(\mathbf{T}^{-1})^{\mathrm{T}}$，其中$\vec{n}$为原曲面的法线向量。矩阵$(\mathbf{T}^{-1})^{\mathrm{T}}$叫做T的逆转置矩阵。你应该认识到，即使$\vec{n}$是单位长度的，向量$\vec{m} = \vec{n}(\mathbf{T}^{-1})^{\mathrm{T}}$也并不一定是单位长度的，因此，对于需要单位长度法线的应用，就需要对 $\vec{m}$ 进行规整化。

推荐的阅读材料

完整而透彻地理解向量代数对计算机图形学程序员来说是非常有用的。可是，在数量庞大的图形学书籍中，内容完整的很少，而且在计算机科学的课程中，也很少讲到这部分内容。具有机械工程或物理专业背景的程序员可能会接触到更多的这方面的材料，虽然他们只是在大学数学课程中学到过相关的内容，而没有学过专门探讨向量代数的课程。与过去相比，现在更是如此，因为在书籍搜索数据库中，有许多以“向量分析”作为标题或标题中包含“向量分析”的课本都已被标为“已售完”。Goldman（1987）引用了如下的两本书：

E.B. Wilson, *Vector Analysis*, Yale University Press, New Haven, CT, 1958.

A. P. Wills, *Vector Analysis with an Introduction to Tensor Analysis*, Dover Publications, New York, 1958.

其他几种有用数学资源为：

Murray Spiegel, *Schaum's Outline of Theory and Problems of Vector Analysis, and an Introduction to Tensor Analysis*, McGraw-Hill, New York, 1959.

Banesh Hoffman, *About Vectors*, Dover, Mineola, NY, 1966, 1975.

Harry Davis and Arthur Snider, *Introduction to Vector Analysis*, McGraw-Hill, New York, 1995.

更多相关的计算机图形学材料包括：

Ronald Goldman, "Vector Geometry: A Coordinate-Free Approach," in *1987 SIGGRAPH Course Notes 19: Geometry for Computer Graphics and Computer Aided Design*, ACM, New York, 1987.

Ronald Goldman, "Illicit Expressions in Vector Algebra," in *ACM Transactions on Graphics*, Vol. 4, No. 3, July 1985.

Tony D. DeRose, "A Coordinate-Free Approach to Geometric Programming," *Math for SIGGRAPH: Course Notes 23, SIGGRAPH '89*, pages 55–115, July 1989.

Tony D. DeRose, *Three-Dimensional Computer Graphics: A Coordinate-Free Approach*. Unpublished manuscript, University of Washington, 1992 (*www.cs .washington.edu*).

James R. Miller, "Vector Geometry for Computer Graphics," *IEEE Computer Graphics and Applications*, Vol. 19, No. 3, May 1999.

James R. Miller, "Applications of Vector Geometry for Robustness and Speed," *IEEE Computer Graphics and Applications*, Vol. 19, No. 4, July 1999.

第 5 章　二维几何图元

本章介绍在应用中常见的几种二维几何图元的定义。有的图元具有多种表示形式。在与对象相关的几何操作中，有的表示方法比其他的表示法更加有效。关于操作的讨论将说明哪一种表示法更合适。

在多边形这类对象的几何操作中，对象可以被看成是一维对象或二维对象。例如，三角形作为一维对象就是封闭的折线边界。作为二维对象，三角形表示其折线边界和它所包围的区域。有的对象在不同的表示中有不同的名称。例如，圆表示一维曲线，而圆盘表示这种曲线和其所包围的区域。如果必要，这种区别将被清楚地标识出来。在没有清楚的名称时，我们将使用立体（solid）一词。例如，在计算点与三角形之间的距离时，我们将三角形看成一个立体。如果点在三角形内，则该距离为零。

5.1　线形对象

线形对象可以被隐含或参数化地表示。对于直线来说，这两种表示法具有相同的效果，但是，我们会看到，在表示射线和线段时，参数形式更方便一些。

5.1.1　隐含形式

直线可以定义为$\vec{n} \cdot X = d$。该直线的法线为$\vec{n} = (n_0, n_1)$，并且直线上的点用变量$X = (x_0, x_1)$来表示。如果P是直线上的一个指定点，那么该直线的方程为$\vec{n} \cdot (X - P) = 0 = \vec{n} \cdot X - d$，其中$\vec{n} \cdot P = d$。这种直线定义叫做法线形式。该直线的方向向量是$\vec{d} = (d_0, d_1) = (n_1, -n_0)$。图 5.1（a）示意了平面上的一条典型的直线。当然，我们无法画出在两个方向上无限延伸的直线，在图中，其中的箭头表示向两边无限延伸。一条直线将平面划分为两部分。直线的法线所在的半平面叫做直线的正面，其代数表示为$\vec{n} \cdot X - d > 0$。另一个半平面叫做直线的负面，其代数表示为$\vec{n} \cdot X - d < 0$。

虽然并不要求$\vec{n}$为单位长度向量，但如果它是单位长度的，那么在许多操作中会很方便。此时$\hat{d}$也是单位长度的。点P和单位长度向量$\hat{d}$与$\hat{n}$构成一个右手坐标系统，其中 P 是原点，而两个单位长度向量表示坐标轴的方向。参见 3.3.3 节中关于坐标系的讨论。任何点X都可以用$X = P + y_0\hat{d} + y_1\hat{n}$来表示，其中$y_0 = \hat{d} \cdot (X - P)$并且$y_1 = \hat{n} \cdot (X - P)$。直线的正面由$y_1 > 0$来表示，直线的负面由$y_1 < 0$来表示，而$y_1 = 0$表示直线本身。

直线的另一种常见的隐含表示形式为

$$ax + by + c = 0$$

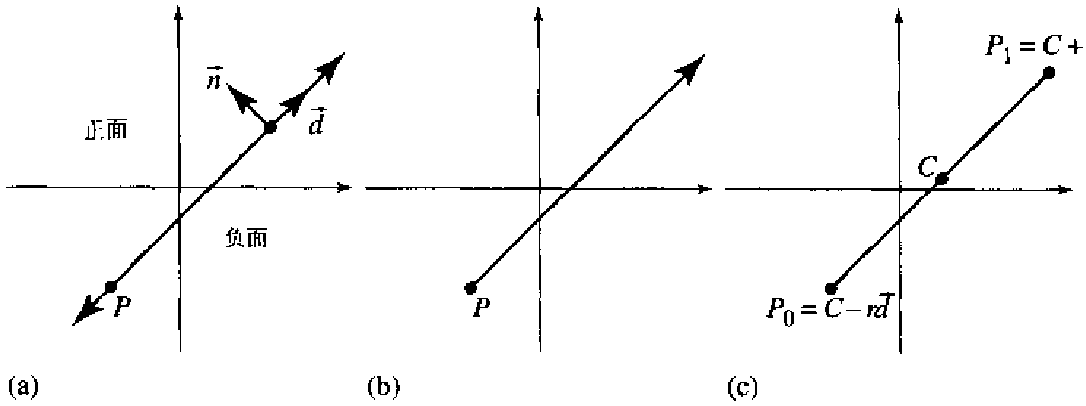


图 5.1　（a）直线；（b）射线；（c）线段

如果我们设$\hat{n} = [a \quad b]$，$X = [x \quad y]$且$d = -c$，则这种形式将等价于前一种形式。如果$a^2 + b^2 = 1$，那么就说直线方程被规整化。一个非规整化的直线方程可以通过在方程两边乘以下式而被规整化：

$$\frac{1}{\sqrt{a^2 + b^2}}$$

这样就相当于另一种表示形式的$\|\hat{n}\| = 1$。将方程规整化后，我们可以很简单地得到一组直观的系数：

$$a = \cos\alpha$$

$$b = \cos\beta$$

$$c = \|\vec{r}\|$$

换句话说，a 和 b 就是垂直于直线的向量（即$\hat{n}$）的 x 和 y 分量，而 c 就是直线到原点的最小（有符号）距离，如图 5.2 所示。

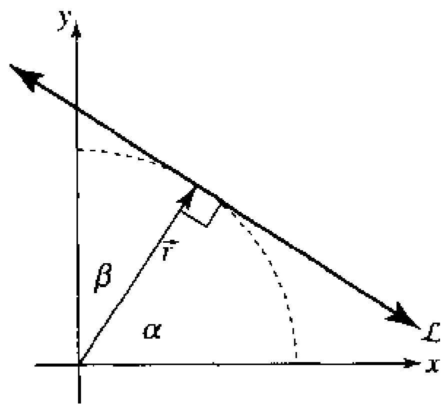


图 5.2　直线的隐含定义

5.1.2　参数形式

直线的参数形式（parametric form）为$X(t) = P + t\vec{d}$，$t \in \mathbb{R}$。射线就是具有参数限制$t \geqslant 0$ 的直线。射线的原点为 P。图 5.1（b）显示了在平面内的射线。正如画直线一样，不

可能将射线画成无限延伸的，因此我们用箭头来表示射线继续无限延伸的方向。直线段，或简称线段，就是具有参数限制 $t \in [t_0, t_1]$ 的直线。如果 P_0 和 P_1 为线段的端点，则线段的标准形式为 $X(t) = (1-t)\ P_0 + tP_1$，$t \in [0, 1]$。这种形式通过设 $\vec{d} = P_1 - P_0$ 而转化为参数形式。线段的对称形式包含一个终点 C，一个单位长度向量 $\hat{d}$ 和一个半径 r。参数表示为 $X(t) = C + t\hat{d}$，$|t| \leqslant r$。对于标准形式而言，线段的长度为 $\|P_1 - P_0\|$；对于对称形式来说，长度为 $2r$。图 5.1（c）显示了在平面内的线段。有时使用 $\langle P_0, P_1 \rangle$ 来表示线段是很方便的。

贯穿本书，术语线形对象（linear component）用来表示直线、射线或线段。

5.1.3 表示法之间的转换

本书的一些算法使用了隐含形式，而另一些算法采用了参数形式。一般这种选择并不是随意的，有些问题采用一种形式比采用另一种形式更易于求解。这里我们介绍如何在这两种形式之间进行转化，以便于你选择最合适的形式。

1. 参数形式到隐含形式的转化

对于参数形式的直线

$$x = P_x + td_x$$
$$y = P_y + td_y$$

其等价的隐含形式为

$$-d_y x + d_x y + (P_x d_y - P_y d_x) = 0$$

2. 隐含形式到参数形式的转化

对于隐含形式的直线

$$ax + by + c = 0$$

其等价的参数形式为

$$P = \begin{bmatrix} \dfrac{-ac}{a^2+b^2} & \dfrac{-bc}{a^2+b^2} \end{bmatrix}$$
$$\vec{d} = [-b \quad a]$$

5.2 三角形

三角形（triangle）由不共线的三个点 P_0，P_1 和 P_2 所确定。如果将 P_0 确认为原点，则三角形具有边向量 $\vec{e}_0 = P_1 - P_0$ 和 $\vec{e}_1 = P_2 - P_0$。其中每一个点都称为三角形的顶点。在大部分应用中，顶点的次序是重要的。如果 P_2 在方向为 $P_1 - P_0$ 的直线的左边，则次序为逆时针的；如果 P_2 在方向为 $P_1 - P_0$ 的直线的右边，则次序为顺时针的。设 $P_i = (x_i, y_i)$，定义

$$\delta = \det\begin{bmatrix} 1 & 1 & 1 \\ x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \end{bmatrix}$$

如果$\delta > 0$，则三角形是逆时针次序的；如果$\delta < 0$，则三角形是顺时针次序的。如果$\delta = 0$，则三角形退化，因为此时三个顶点共线。图 5.3 显示了两个具有不同次序的三角形。在本书中，三角形都采用逆时针次序。观察一下，如果你沿着三角形逆时针方向行走，三角形所包围的区域总是在你的左边。三角形的三点表示法叫做顶点形式。

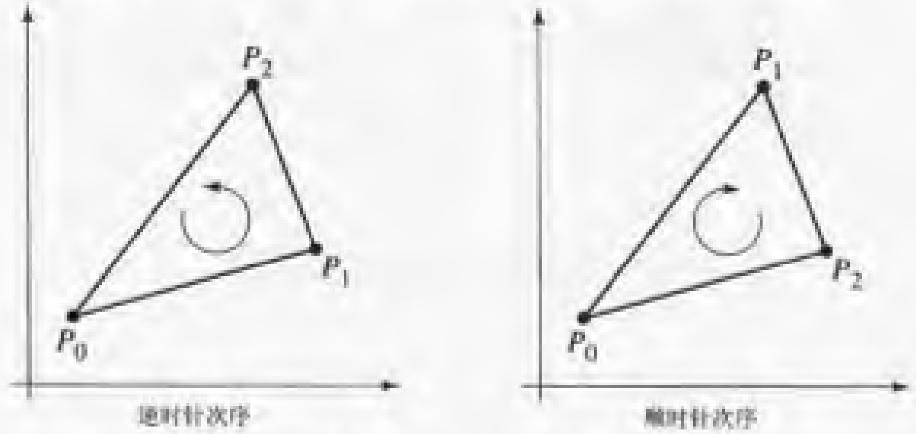


图 5.3 三角形的两种可能次序

三角形的参数形式为$X(t_0, t_1) = P_0 + t_0\vec{e}_0 + t_1\vec{e}_1$, $t_0 \in [0, 1]$，$t_1 \in [0, 1]$，并且$0 \leqslant t_0 + t_1 \leqslant 1$。三角形的重心形式为$X(c_0, c_1, c_2) = c_0P_0 + c_1P_1 + c_2P_2$，对所有$i$, $c_i \in [0, 1]$并且$c_0 + c_1 + c_2 = 1$。参数形式是一个方程$X : D \subset \mathbb{R}^2 \rightarrow \mathbb{R}^2$，其定义域$D$为平面上的等腰直角三角形，其值域为三个指定的顶点所确定的三角形。图 5.4 显示了定义域和值域三角形，以及它们与顶点之间的对应关系。重心形式是一个方程$X : D \subset \mathbb{R}^3 \rightarrow \mathbb{R}^2$，其定义域$D$为平面上的等腰三角形，其值域为三个指定的顶点所确定的三角形。图 5.5 显示了定义域和值域三角形，以及它们与顶点之间的对应关系。

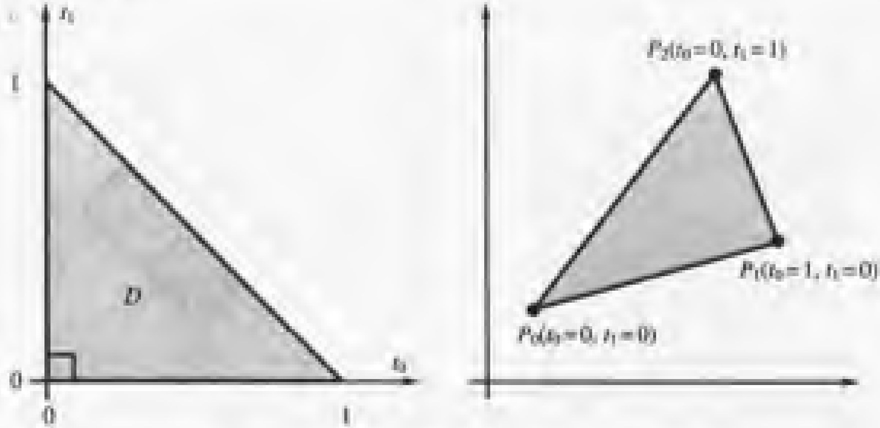


图 5.4 三角形的参数形式的定义域和值域

如果$P_i = (x_i, y_i)$，其中$0 \leqslant i \leqslant 2$，那么$\vec{e}_0 = (x_1 - x_0, y_1 - y_0)$且$\vec{e}_1 = (x_2 - x_0, y_2 - y_0)$。三角形的有符号面积就是前面所提到的行列式，其符号与三角形顶点的次序相关：

$$\text{Area}(P_0, P_1, P_2) = \frac{1}{2}\det\begin{bmatrix} 1 & 1 & 1 \\ x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \end{bmatrix} = \frac{1}{2}((x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0))$$

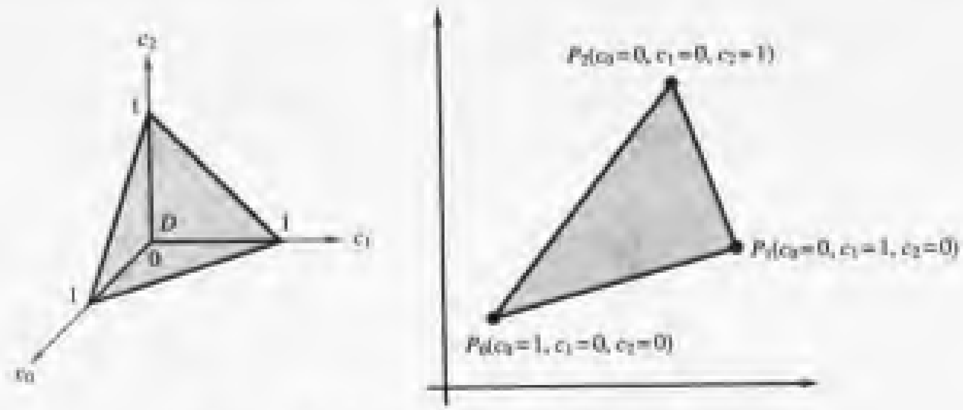


图 5.5　三角形的重心形式的定义域和值域

5.3　矩形

矩形（rectangle）由一个点 P 和两条互相垂直的边 $\vec{e}_0$ 和 $\vec{e}_1$ 所定义。这种形式叫做顶点一边形式。矩形的参数形式为 $X(t_0, t_1) = P + t_0\vec{e}_0 + t_1\vec{e}_1$，其中 $t_0 \in [0, 1]$，$t_1 \in [0, 1]$。如果矩形的边向量平行于坐标轴，那么矩形是轴对齐（axis aligned）的。虽然所有矩形都可认为是有向的（oriented），但是这一术语通常用来强调矩形并不一定是轴对齐的。矩形的对称形式（symmetric form）包含一个中心点 C，两个互相垂直的单位长度向量 $\hat{u}_0$ 和 $\hat{u}_1$，以及两个长度 $e_0 > 0$ 和 $e_1 > 0$。其参数形式为 $X(t_0, t_1) = C + t_0\hat{u}_0 + t_1\hat{u}_1$，其中 $|t_0| \leqslant e_0$ 且 $|t_1| \leqslant e_1$。对于参数形式，矩形的面积为 $\|\vec{e}_0\|\ \|\vec{e}_1\|$；对于对称形式，矩形的面积为 $4e_0e_1$。图 5.6 显示了定义域正方形，值域矩形，以及参数形式中各顶点之间的相互关系。图 5.7 显示了矩形的对称形式。

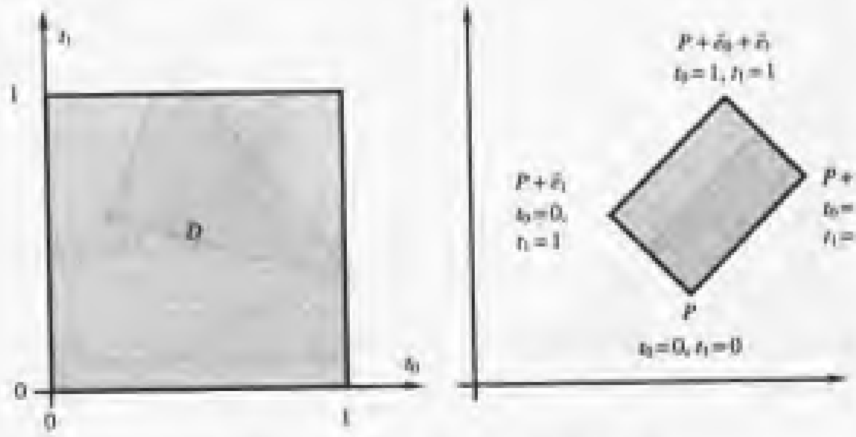


图 5.6　矩形参数形式的定义域和值域

5.4　折线和多边形

折线（polyline）包含有限数量的线段 $\langle P_i, P_{i+1}\rangle$，其中 $0 \leqslant i < n$。相邻的线段共用一个

端点。尽管在应用中不常见，但该定义可被扩展，以允许折线包含射线和直线。一个例子是如下的折线，它包含端点为(0, 0)和 (0, 1)的线段、原点为(0, 0)、方向向量为 (1, 0)的射线，以及原点为 (0, 1)、方向向量为 (−1, 0)的射线。图 5.8 显示了平面上的典型折线。如果线的最后一点与第一点通过一条线段相连在一起，则折线是闭合的。本书的表示方式是指定多余的一点 $P_n = P_0$，这是为了便于索引。不闭合的折线被称为开放的。

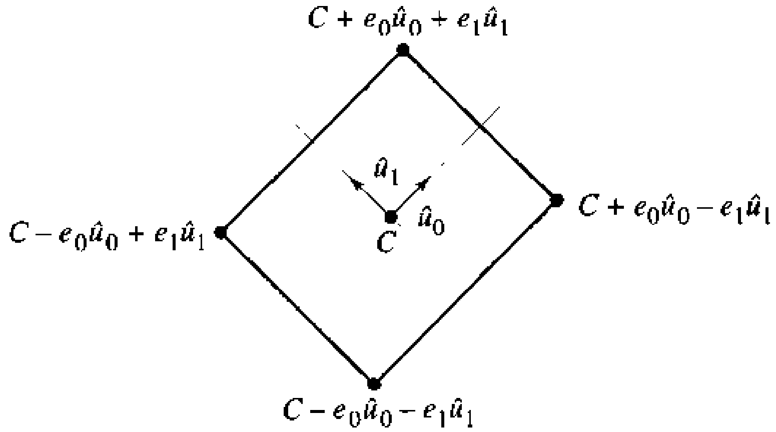


图 5.7 矩形的对称形式

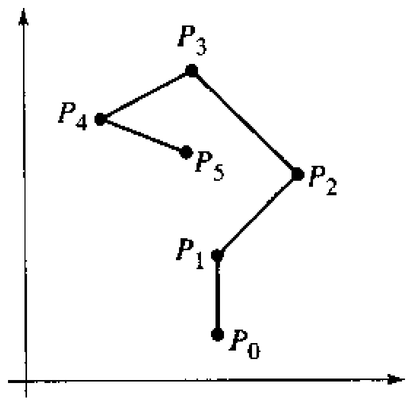


图 5.8 一条典型的折线

多边形（polygon）是闭合的折线。每一个点 P_i 叫做多边形的顶点（vertex），每一条线段叫做多边形的边。如果不相邻的边不相交，则多边形是简单多边形。简单多边形包围平面内的一个简单相连的区域。该区域内的点被称为在多边形内。简单多边形的顶点次序与三角形的顶点次序一样，可分为逆时针次序和顺时针次序。如果沿顶点行走保持区域在左边，则顶点次序是逆时针的。如果对于多边形内的任意两点，连接着这两点的线段也在多边形内，则多边形是凸多边形（convex）。特殊的凸多边形有三角形、矩形、平行四边形（四条边两两平行）和凸四边形（具有四条边，每一顶点都在其余三个顶点构成的三角形之外）。非凸多边形被称为凹多边形（concave）。

图 5.9 显示了两个简单多边形。图 5.9（a）中的多边形是一个凹多边形，因为连接两个内点的线段并不完全在多边形内。图 5.9（b）中的多边形是一个凸多边形，因为不论如何选取多边形内的两点，连接它们的线段总是在多边形内。图 5.10 显示了两个非简单多边

形。图 5.10（a）中的多边形有两条不相邻的边(P_1, P_2)和(P_3, P_4)相交，交点不是多边形的顶点。图 5.10（b）是同一个多边形，只是交点为一个顶点，但是多边形依然是非简单的，因为它的多条不相邻线段相交于同一点。后一种类型的多边形叫做多体（polysolids）（Maynard 和 Tavernini，1984）。

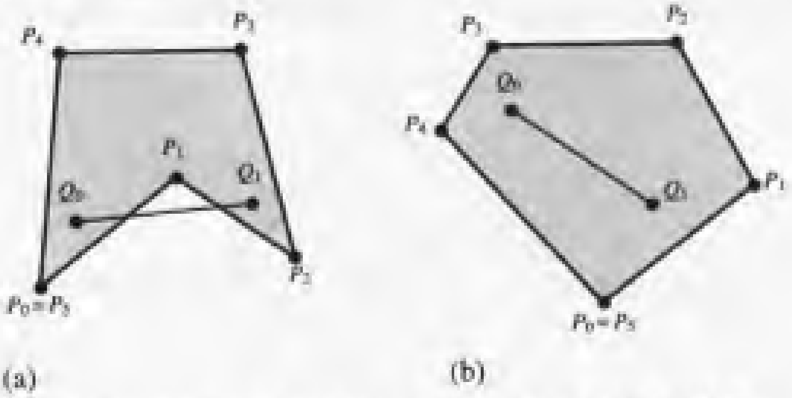


图 5.9　示例：（a）简单的凹多边形；（b）简单的凸多边形

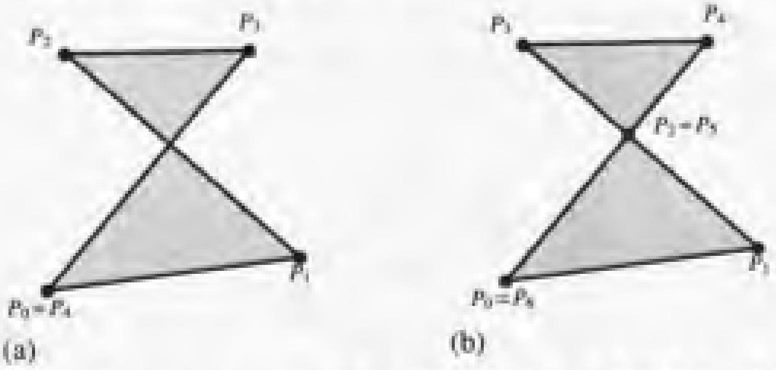


图 5.10　非简单多边形的示例：（a）交集不是顶点；（b）交集是顶点，多边形是多体

折线链是不相邻的线段不相交的开放折线。折线链C是关于直线$\mathcal{L}$严格单调的，如果垂直于$\mathcal{L}$的每一条线与C的交点最多只有一点。如果垂直于$\mathcal{L}$的任一条线与C的交点为空，一个点或一条线段，则折线链C是单调的。简单的多边形不可能是单调的折线链。单调多边形是能划分为两个单调折线链的简单多边形。图 5.11 显示了一条严格单调的折线链和一条单调的折线链。图 5.12 显示了一个单调多边形。

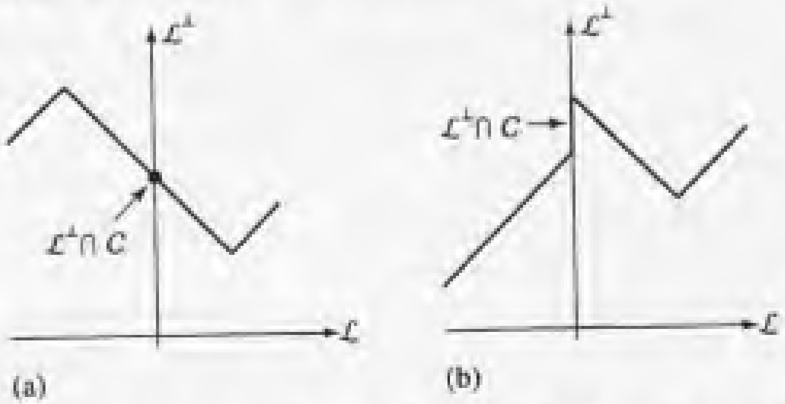


图 5.11　折线链的例子：（a）严格单调；（b）单调，但不严格单调

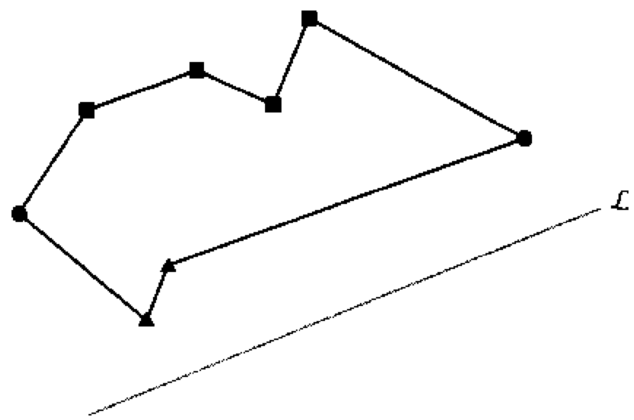


图 5.12　一个单调多边形。正方形是一条链的顶点集；三角形是另一条链的顶点集；圆是这两条链的顶点集

5.5　二次曲线

二次曲线（quadratic curves）由一般的具有两个变量的二次方程所隐含确定，即

$$a_{00}x_0^2 + 2a_{01}x_0x_1 + a_{11}x_1^2 + b_0x_0 + b_1x_1 + c = 0 \tag{5.1}$$

设 $\mathbf{A} = [a_{ij}]$ 为对称的 2×2 矩阵，并设 $B = [b_i]$ 和 $X = [x_i]$ 为 2×1 向量。二次曲线方程的矩阵形式为

$$X^{\mathrm{T}}\mathbf{A}X + B^{\mathrm{T}}X + c = 0 \tag{5.2}$$

二次曲线方程可定义点、线、圆、椭圆、抛物线或双曲线。这种方程也可能无解。

通过提取因子 A 并改变变量，可以更简单地确定方程（5.2）所定义的对象。由于 A 是对称矩阵，因此它可被分解为 $\mathbf{A} = \mathbf{R}^{\mathrm{T}}\mathbf{D}\mathbf{R}$，其中 R 为旋转矩阵，它的各行为 A 的特征向量；D 是对角线矩阵，其对角线各项为 A 的特征值。为了提取因子 A，参见 A.3 节中的特征值分解的内容。定义 $E = \mathbf{R}B$ 和 $Y = \mathbf{R}X$。方程（5.2）变换为

$$Y^{\mathrm{T}}\mathbf{D}Y + E^{\mathrm{T}}Y + c = d_0y_0^2 + d_1y_1^2 + e_0y_0 + e_1y_1 + c = 0 \tag{5.3}$$

根据 D 的对角线元素来进行分类。如果一个对角线元素 d_i 不为零，那么方程（5.3）中对应的项 y_i 和 y_i^2 可通过完全平方式而被因式分解。例如，如果 $d_0 \neq 0$，那么

$$\begin{aligned} d_0y_0^2 + e_0y_0 &= d_0\left(y_0^2 + \frac{e_0}{d_0}y_0\right) \\ &= d_0\left(y_0^2 + \frac{e_0}{d_0}y_0 + \frac{e_0^2}{4d_0^2} - \frac{e_0^2}{4d_0^2}\right) \\ &= d_0\left(y_0 + \frac{e_0}{2d_0}\right)^2 - \left(\frac{e_0}{2d_0}\right)^2 \end{aligned}$$

【情形 1】 $d_0 \neq 0$ 且 $d_1 \neq 0$。方程因式分解为

$$d_0\left(y_0 + \frac{e_0}{2d_0}\right)^2 + d_1\left(y_1 + \frac{e_1}{2d_1}\right)^2 = \frac{e_0^2}{4d_0} + \frac{e_1^2}{4d_1} - c =: r$$

假定 $d_0d_1>0$。当 $d_0r<0$ 时，没有实数解。当 $r=0$ 时，解为一个点。如果 $d_0r>0$，则当 $d_0\neq d_1$ 时，解为一个椭圆；当 $d_0=d_1$ 时，解为一个圆。假定 $d_0d_1<0$。如果 $r\neq 0$，则解为一条双曲线。如果 $r=0$，则解为两条相交的线，这是三维空间中的锥体在二维空间中的等价体。图 5.13 显示了各种可能的情形。■

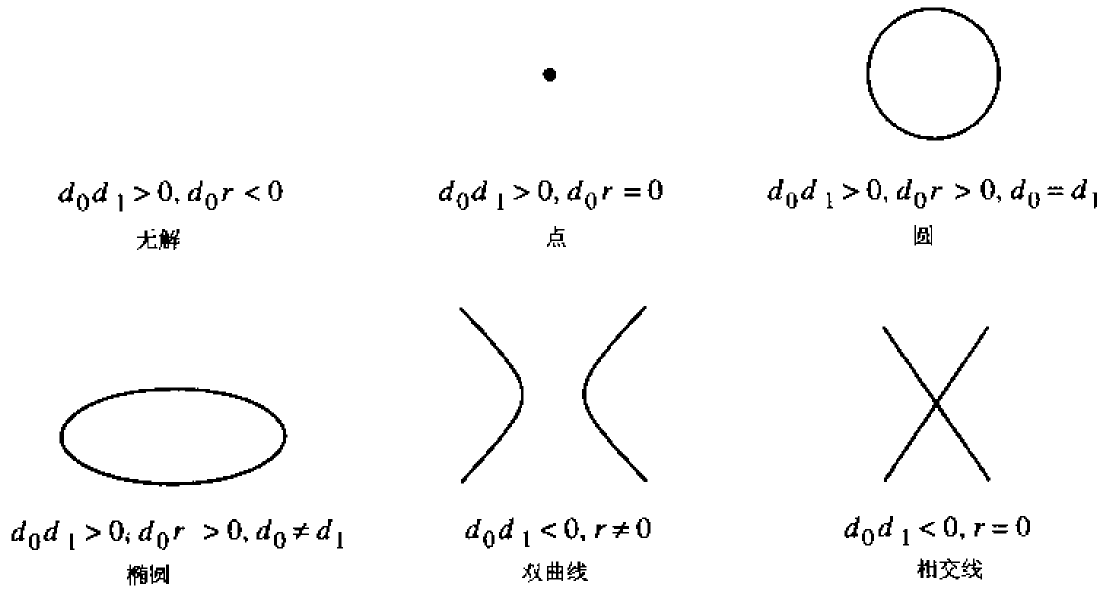


图 5.13　二次方程的解，取决于 $d_0\neq 0$，$d_1\neq 0$ 和 r 的值

【情形 2】 $d_0\neq 0$ 且 $d_1=0$。方程因式分解为

$$d_0\left(y_0+\frac{e_0}{2d_0}\right)^2+e_1y_1=\frac{e_0^2}{4d_0}-c=:r$$

如果 $e_1=0$，则存在三种情形。当 $d_0r<0$ 时，没有实数解。当 $r=0$ 时，解为一条直线。如果 $d_0r>0$，解为两条平行直线，这是三维空间中的柱体在二维空间中的等价体。如果 $e_1\neq 0$，解为一条抛物线。图 5.14 显示了各种可能的情形。■

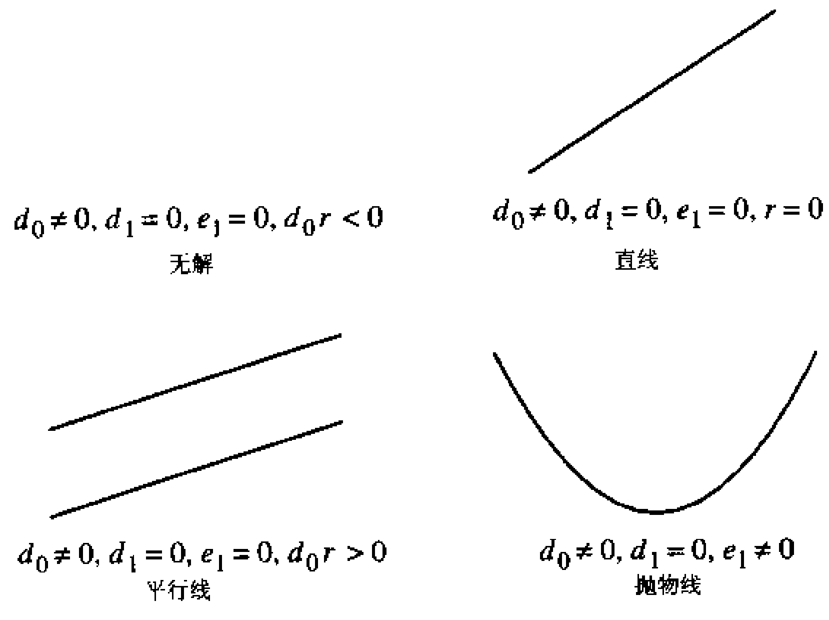


图 5.14　二次方程的解，取决于 $d_0\neq 0$，$d_1=0$，e_1 和 r 的值

【情形 3】 $d_0=0$ 且 $d_1\neq 0$。与 $d_0\neq 0$ 且 $d_1=0$ 的情形对称一致。■

【情形 4】 $d_0=0$ 且 $d_1=0$。方程为

$$e_0y_0+e_1y_1+c=0$$

如果 $e_0=e_1=0$，则当 $c\neq 0$ 时，无解；当 $c=0$ 时，方程变为 $0=0$。如果 $e_0\neq 0$ 或 $e_1\neq 0$，则解为一条直线。■

5.5.1 圆

圆（circle）包括圆心 C 和半径 $r>0$。圆的距离形式为 $\|X-C\|=r$。其参数形式为 $X(t)=C+\ r\hat{u}(t)$，其中 $\hat{u}(t)=(\cos t,\sin t)$，$t\in[0,2\pi)$。为了检验这一方程，观察 $\|X(t)-C\|=\|r\hat{u}(t)\|\ =r\|\hat{u}(t)\|=r$，其中最后一个等式是正确的，因为 $\hat{u}(t)$ 是一个单位向量。图 5.15 显示了（隐含的）距离形式和参数形式。其二次曲线形式是 $X^{\mathrm{T}}\mathrm{I}X+B\cdot X+c=0$，其中I为 2×2 单位矩阵。在二次曲线形式中，系数通过 $B=-2C$ 和 $c=C^{\mathrm{T}}C-r^2$ 而与圆心和半径相关。

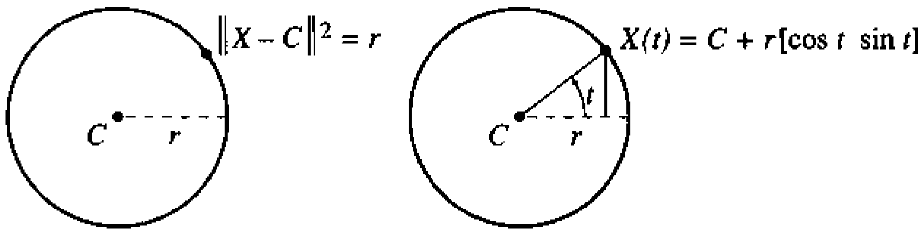


图 5.15　用距离（隐含）定义的圆及其参数形式

对于距离形式和参数形式，圆的面积为 πr^2；而对于二次曲线形式，圆的面积为 $\pi(B^{\mathrm{T}}B/4-c)$。

5.5.2 椭圆

椭圆（ellipse）包括椭圆心 C，两个轴半径 $\ell_0>0$ 和 $\ell_1>0$，以及一个相对 x 轴的逆时针方向的角度 θ，如图 5.16 所示。设 $\mathrm{D}=\mathrm{Diag}(1/\ell_0^2,1/\ell_1^2)$ 且 $\mathrm{R}=R(\theta)$ 为与指定角度对应的旋转矩阵。椭圆的因子形式为 $(X-C)^{\mathrm{T}}\mathrm{R}^{\mathrm{T}}\mathrm{D}\mathrm{R}(X-C)=1$。椭圆的参数形式为 $X(t)=C+\mathrm{R}^{\mathrm{T}}\mathrm{D}^{-1/2}\ \hat{u}(t)$，其中 $\hat{u}(t)=(\cos t,\sin t)$，$t\in[0,2\pi)$。为了检验这一方程，观察因子形式说明 $\|\mathrm{D}^{1/2}\mathrm{R}(X\ \ -C)\|=1$。该向量的长度表明它是一个单位长度，因此 $\mathrm{D}^{1/2}\mathrm{R}(X-C)=\hat{u}(t)$ 是合理的选择。求解 X 将得到参数形式。其二次曲线形式是 $X^{\mathrm{T}}\mathrm{A}X+B\cdot X+c=0$，其中 A 为 2×2 矩阵，其对角线各项为正，而且其行列式也为正。并且，$C=-\mathrm{A}^{-1}B/2$，以及 $\mathrm{R}^{\mathrm{T}}\mathrm{D}\mathrm{R}=\mathrm{A}/(B^{\mathrm{T}}\mathrm{A}^{-1}B/4\ \ -c)$。

对于因子形式和参数形式，椭圆的面积为 $\pi\ell_0\ell_1$；而对于二次曲线形式，圆的面积为 $\pi(B^{\mathrm{T}}\mathrm{A}^{-1}B/4-c)/\sqrt{\det(\mathrm{A})}$。

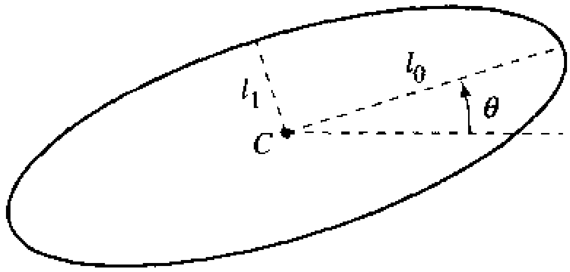


图 5.16　椭圆的定义

5.6　多项式曲线

平面上的多项式曲线（polynomial curve）是一个具有向量值的方程 $X: D \subset \mathbb{R} \to R \subset \mathbb{R}^2$，设为 $X(t)$，其定义域为 D，值域为 R。$X(t)$ 的分量 $X_i(t)$ 为如下指定参数的多项式

$$X_i(t) = \sum_{j=0}^{n_i} a_{ij} t^j$$

其中 n_i 为多项式的次数。在许多应用中，分量的次数是相同的，此时，曲线方程记为 $X(t) = \sum_{j=0}^{n} A_j t^j$，$A_j \in \mathbb{R}^2$ 为已知点。即使次数不同，也可以使用向量标记，设 $n = \max_i n_i$，设系数 $a_{ij} = 0$ 对 $n_i < j \leqslant n$ 成立。应用程序中的定义域 D 一般为 $\mathbb{R}$ 或 $[0,1]$。一个有理多项式曲线是一个具有向量值的方程 $X(t)$，其 $X_i(t)$ 为多项式之比

$$X_i(t) = \frac{\sum_{j=0}^{n_i} a_{ij} t^j}{\sum_{j=0}^{m_i} b_{ij} t^j}$$

其中 n_i 和 m_i 分别为分子和分母多项式的次数。

贝塞尔曲线、B 样条曲线和非均衡有理 B 样条（NURBS）曲线是计算机图形学中常见的几种二次曲线。我们在此只给出这些曲线的定义。它们的各种有趣的性质可以从其他关于曲线和曲面的书籍中找到。

5.6.1　贝塞尔曲线

平面贝塞尔曲线（Bézier curve）由一组点 $P_i \in \mathbb{R}^2$，$0 \leqslant i \leqslant n$ 构成，这组点称为控制点，其方程为

$$X(t) = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} P_i = \sum_{i=0}^{n} B_i(t) P_i, \quad t \in [0,1]$$

其中

$$\binom{n}{i} = \frac{n!}{i!(n-i)!}$$

为从一组 n 项中选出 i 项的组合数目。实数多项式 $B_i(t)$ 叫做伯恩斯坦多项式，每一项的次数都为 n。因此，$X(t)$ 的多项式分量的次数也为 n。图 5.17 显示了一条三次贝塞尔曲线，

以及其控制点和控制多项式。

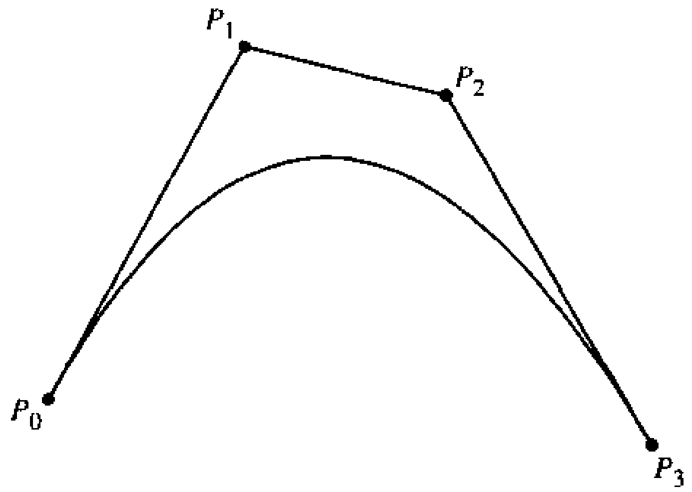


图 5.17　三次贝塞尔曲线

5.6.2　B 样条曲线

一条度数为 j 的平面 B 样条曲线（B-spline curve）由一组点（称为控制点）$P_i \in \mathbb{R}^2$ 和一组单调参数（称为结）$t_i\ (t_i \leqslant t_{i+1})$构成，其中 $0 \leqslant i \leqslant n$，其方程为

$$X(t) = \sum_{i=0}^{n} B_{i,j}(t) P_i$$

其中 $t \in [t_0, t_n]$ 且 $1 \leqslant j \leqslant n$。向量$(t_0, \ldots, t_n)$叫做结向量（knot vector）。实数多项式 $B_{i,j}(t)$ 的次数为 j，由考克斯德布尔递归公式定义

$$B_{i,0}(t) = \begin{cases} 1, & t_i \leqslant t < t_{i+1} \\ 0, & \text{否则} \end{cases}$$

且

$$B_{i,j}(t) = \frac{(t - t_i) B_{i,j-1}(t)}{t_{i+j-1} - t_i} + \frac{(t_{i+j} - t) B_{i+1,j-1}(t)}{t_{i+j} - t_{i+1}}$$

其中 $1 \leqslant j \leqslant n$。$X(t)$ 的多项式分量实际上分段地定义在区间 $[t_i, t_{i+1}]$ 上。在每一个区间上，多项式的次数都为 j。结的值并不要求是均匀分布的。此时的 B 样条曲线叫做非均衡 B 样条曲线。如果结的值均匀分布，则叫做均衡 B 样条曲线。图 5.18 显示了一条均衡的三次 B 样条曲线，以及其控制点和控制多项式。

5.6.3　非均匀有理 B 样条曲线

平面非均匀有理 B 样条曲线（nonuniform B-spline curve）或 NURBS 曲线可从三维空间中的非均衡 B 样条多项式曲线中得到。其控制点为$(P_i, 1)$，其中 $0 \leqslant i \leqslant n$，其权重为 $w_i > 0$，其多项式曲线为

$$(Y(t), w(t)) = \sum_{i=0}^{n} B_{i,j}(t) w_i (P_i, 1)$$

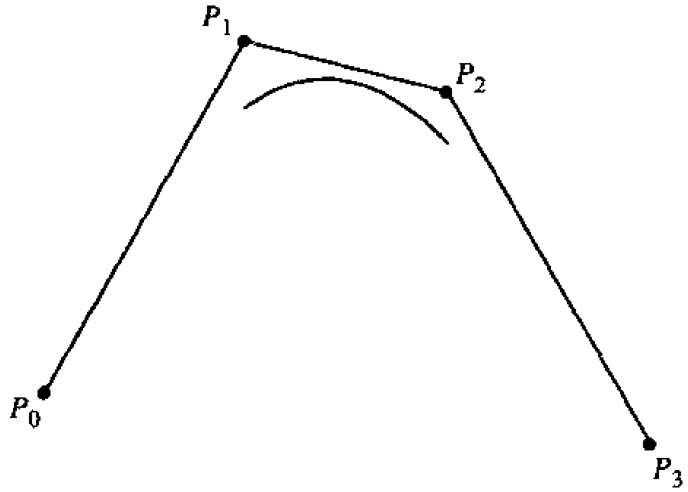


图 5.18　三次 B 样条曲线

其中 $B_{i,j}(t)$ 为与在上一节中定义的多项式相同的多项式。可以通过如下的方法得到 NURBS 曲线，即将 $(Y(t), w(t))$ 处理为均匀向量，并通过最后一个分量（一般叫做权重）来进行分解，以得到三维空间中的一个投影

$$X(t) = \frac{Y(t)}{w(t)} = \sum_{i=0}^{n} R_{i,j}(t) P_i$$

其中

$$R_{i,j}(t) = \frac{w_i B_{i,j}(t)}{\sum_{k=0}^{n} w_k B_{k,j}(t)}$$

第 6 章　二维距离

本章介绍计算二维空间中几何图元之间的距离的有关知识。应用程序可能不愿花太大的代价来计算平方根，因此本章的许多算法都仅提供计算距离平方的方法。当然，任何距离算法的基础都是两点$X=(x_0,x_1)$和$Y=(y_0,y_1)$之间的距离平方

$$\text{Distance}^2(X,Y)=\|X-Y\|^2=(x_0-x_1)^2+(y_0-y_1)^2 \tag{6.1}$$

我们将首先讨论计算点与其他对象之间的距离的算法；在本章的后半部分将讨论其他的组合。当两个对象都是折线或多边形边界的凸形（包括一个是线形对象的简化情形）时，计算距离的算法可通过计算点对象的距离公式求得的值中的极小值来实现。例如，线段与三角形之间的距离可通过求取一个一维函数的极小值来实现。如果$F(X,T)$是点 X 与三角形 T 之间距离的平方，那么线段$X(t)=P_0+t(P_1-P_0)$（$t\in[0,1]$）和三角形之间的距离为$G(t)=F(X(t),T)$。可计算$G(t)$（$t\in[0,1]$）的极小值。这样的一种迭代方法当然能产生合理的距离平方的估计数值，但是，与不用迭代的距离平方的计算方法相比，一般这种方法将消耗更多的时间来找到估计数值。中庸的算法是既要易于实现，又要有较高的效率。

6.1　点到线形对象的距离

本节将介绍计算点与直线、射线和线段之间的距离的算法，即点－直线、点－射线和点－线段算法。

6.1.1　点到直线的距离

对于一个点 Y 和一条参数形式为 $X(t)=P+t\vec{d}$ 的直线$\mathcal{L}$，直线上与 Y 最近的点是 Y 以某种参数值 $\bar{t}$ 在直线上的投影$X(\bar{t})$。图 6.1 说明了这种关系。如图中所示，向量$Y-X(\bar{t})$必须垂直于直线方向$\vec{d}$。因此

$$0=\vec{d}\cdot(Y-X(\bar{t}))=\vec{d}\cdot(Y-P-\bar{t}\vec{d})=\vec{d}\cdot(Y-P)-\bar{t}\|\vec{d}\|^2$$

并且投影的参数为$\bar{t}=\vec{d}\cdot(Y-P)/\|\vec{d}\|^2$。距离平方为$\|Y-P-\bar{t}\vec{d}\|^2$。通过一些代数运算可得

$$\text{Distance}^2(Y,\mathcal{L})=\|Y-P\|^2-\frac{\left(\vec{d}\cdot(Y-P)\right)^2}{\|\vec{d}\|^2} \tag{6.2}$$

如果 $\vec{d}$ 已经是单位向量，那么方程可简化一些，因为$\|\hat{d}\|=1$，并且计算距离平方时，不需要进行除法运算。

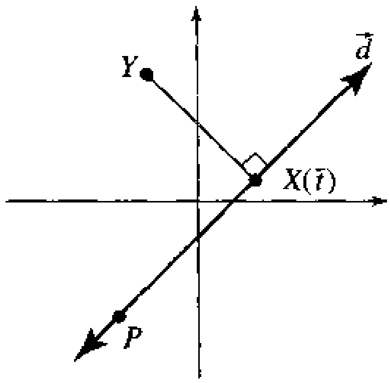


图 6.1　一条直线上距指定点 Y 最近的点 $X(\bar{t})$

如果直线表示为 $\vec{n}\cdot X=c$，那么直线上最近的点 K 对于某些 s 满足关系 $Y=K+s\vec{n}$。方程两边同时点乘 $\vec{n}$ 可得 $\vec{n}\cdot Y=\vec{n}\cdot K+s\|\vec{n}\|^2=c+s\|\vec{n}\|^2$，因此，$s=(\vec{n}\cdot Y-c)/\|\vec{n}\|^2$。点与直线的距离为 $\|Y-K\|=|s|\|\vec{n}\|$，或者

$$\text{Distance}\left(Y,\mathcal{L}\right)=\frac{|\vec{n}\cdot Y-c|}{\|\vec{n}\|} \tag{6.3}$$

如果 $\vec{n}$ 已经是单位向量，那么方程可简化一些，因为 $\|\hat{n}\|=1$，并且计算距离平方时，不需要进行除法运算。

方程（6.3）看上去比方程（6.2）简单一些，而且计算效率也更高一些。假定 $\hat{d}$ 和 $\hat{n}$ 是单位向量，从如下的恒等式中可以很清楚地看出这两个方程之间的关系：

$$\begin{aligned}\|Y-P\|^2&=(Y-P)^{\mathrm{T}}(Y-P)\\&=(Y-P)^{\mathrm{T}}\mathrm{I}(Y-P)\\&=(Y-P)^{\mathrm{T}}(\hat{d}\hat{d}^{\mathrm{T}}+\hat{n}\hat{n}^{\mathrm{T}})(Y-P)\\&=\left(\hat{d}\cdot(Y-P)\right)^2+\left(\hat{n}\cdot(Y-P)\right)^2\\&=\left(\hat{d}\cdot(Y-P)\right)^2+\left(\hat{n}\cdot Y-c\right)^2\end{aligned}$$

建立该恒等式的关键是 $\mathrm{I}=\hat{d}\hat{d}^{\mathrm{T}}+\hat{n}\hat{n}^{\mathrm{T}}$，其中 I 为 2×2 的单位矩阵。该恒等式对任何一对正交向量都成立。其证明依赖于一个事实，即 $\{\hat{d},\hat{n}\}$ 为 $\mathbb{R}^2$ 上的正交基底，因此每一个向量都可表示为

$$\mathrm{I}X=X=(\hat{d}^{\mathrm{T}}X)\hat{d}+(\hat{n}^{\mathrm{T}}X)\hat{n}=(\hat{d}\hat{d}^{\mathrm{T}})X+(\hat{n}\hat{n}^{\mathrm{T}})X=(\hat{d}\hat{d}^{\mathrm{T}}+\hat{n}\hat{n}^{\mathrm{T}})X$$

由于这对所有 X 都成立，因此必然有 $\mathrm{I}=\hat{d}\hat{d}^{\mathrm{T}}+\hat{n}\hat{n}^{\mathrm{T}}$。注意不要将 $\hat{d}\hat{d}^{\mathrm{T}}$ 项与点积 $\hat{d}^{\mathrm{T}}\hat{d}$ 项混淆。向量 $\hat{d}$ 是一个 2×1 向量，因此 $\hat{d}^{\mathrm{T}}$ 是一个 1×2 向量，且乘积 $\hat{d}\hat{d}^{\mathrm{T}}$ 为一个 2×2 矩阵。

6.1.2　点到射线的距离

计算点到射线的距离与计算点到直线的距离相似。不同的是，在有些情况下，Y 在包含射线 $\mathcal{R}$ 的直线上的投影可能不是射线上的一个点。比如，Y 在射线之后。图 6.2 显示了

两种可能。图 6.2（a）显示了投影在射线上的情形。图 6.2（b）显示了投影不在射线上的情形。此时，$\bar{t} < 0$，并且到 Y 最近的点是射线的原点 P。距离平方为

$$\text{Distance}^2(Y, \mathcal{R}) = \begin{cases} \|Y - P\|^2 - \frac{\left(\vec{d}\cdot(Y-P)\right)^2}{\|\vec{d}\|^2}, & \vec{d}\cdot(Y-P) > 0 \\ \|Y - P\|^2, & \vec{d}\cdot(Y-P) \leqslant 0 \end{cases} \tag{6.4}$$

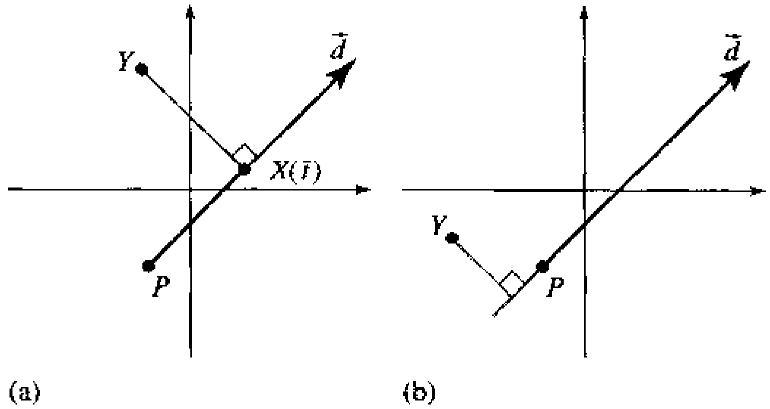


图 6.2 一条射线上距指定点最近的点：（a）$X(\bar{t})$ 距 Y 最近；（b）P 距 Y 最近

6.1.3 点到线段的距离

计算点到线段的距离与计算点到直线的距离相似。不同的是，在有些情况下，Y 在包含线段 S 的直线上的投影可能不是线段上的一个点。投影可能在线段的起点之前或终点之后。图 6.3 显示了三种可能。方向向量为 $\vec{d} = P_1 - P_0$，即线段的两个端点之差。参数区间是[0，1]。值 $\bar{t}$ 是为直线而计算的，但是对于线段来说，需要通过参数区间[0，1]来检验。距离平方为

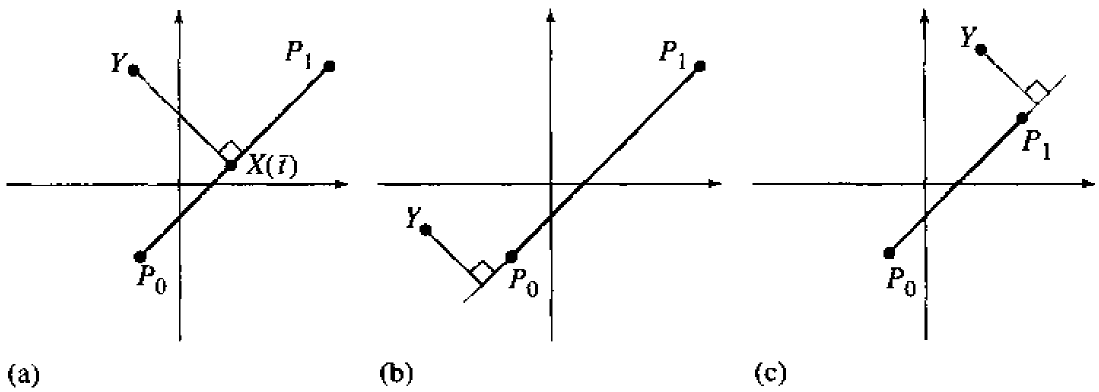


图 6.3 一条线段上距指定点最近的点：（a）$X(\bar{t})$ 距 Y 最近；（b）P_0 距 Y 最近；（c）P_1 距 Y 最近

$$\text{Distance}^2(Y, S) = \begin{cases} \|Y - P_0\|^2, & \bar{t} \leqslant 0 \\ \|Y - (P_0 + \bar{t}\vec{d})\|^2, & \bar{t} \in (0, 1) \\ \|Y - P_1\|^2, & \bar{t} \geqslant 1 \end{cases} \tag{6.5}$$

其中 $\bar{t} = \vec{d}\cdot(Y - P_0)/\|\vec{d}\|^2$。对于对称形式（参见 5.1.2 节），距离平方为

$$\text{Distance}^2(Y,S)=\begin{cases}\|Y-(C-r\vec{d})\|^2, & \bar{t}\leqslant -r\\ \|Y-(C+\bar{t}\vec{d})\|^2, & |\bar{t}|<r\\ \|Y-(C+r\vec{d})\|^2, & \bar{t}\geqslant r\end{cases} \tag{6.6}$$

其中$\bar{t}=\vec{d}\cdot(Y-C)$。

对于需要进行大量的距离计算的应用程序来说，尽可能快地计算距离平方是很重要的。如果$\vec{d}$不是单位向量，那么在公式中将出现除法运算。可以通过空间一时间交换来避免除法运算。如果内存允许，无论是参数形式还是标准形式，都可以先计算数量$1/\|\vec{d}\|^2$，并将其与线段存放在一起。如果内存不允许，那么除法运算可推迟到绝对需要时才进行。例如，在标准形式中，推迟除法运算的算法为

```
float SquaredDistance(Point Y, Segment S)
{
    Point D = S.P1 - S.P0;
    Point YmP0 = Y - S.P0;
    float t = Dot(D, YmP0);

    if (t <= 0) {
        // P0 is closest to Y
        return Dot(YmP0, YmP0);
    }

    float DdD = Dot(D, D);
    if (t >= DdD) {
        // P1 is closest to Y
        Point YmP1 = Y - S.P1;
        return Dot(YmP1, YmP1);
    }

    // closest point is interior to segment
    return Dot(YmP0, YmP0) - t * t / DdD;
}
```

6.2 点到折线的距离

计算点Y与顶点为P_0至P_n、线段为S_i（$0\leqslant i<n$，端点为P_i和P_{i+1}）的折线$\mathcal{L}$之间的距离时，最直接的算法是计算点与折线的各线段之间的距离的最小值

$$\text{Distance}^2(Y,\mathcal{L})=\min_{0\leqslant i<n}\text{Distance}^2(Y,S_i) \tag{6.7}$$

对于具有大量的线段或涉及大量折线并且需要频繁地计算距离的应用程序来说，盲目地迭代搜索线段一般是很费时的。

一种变体算法是采用排除方法，即确定一条线段并不足够接近测试点，不能用来代替当前的最小值μ。这种方法节省时间的地方是避免了当一条线段上最接近Y的点在线段之内时可能的除法运算。设$Y=(a,b)$。只要当前的最小距离保持为μ，那么任何在以Y为圆心、μ为半径的圆之外的线段与Y的距离都比μ大，因此这种线段不能用来更新μ。图6.4说明了这种情况。完整的距离计算将排除线段S_1和S_2，因为它们在以Y为圆心、μ为半

径的圆之外。线段 S_3 没有被排除，因为它与圆相交。然而，这种方法也将带来问题，因为排除测试要求计算线段与 Y 的距离，而这正是我们试图避免的计算。

一种更快但也更粗糙的方法是使用包含圆的轴对齐无限带来进行排除测试。设 $S_i=\langle(x_i,y_i),(x_{i+1},y_{i+1})\rangle$ 为将要测试的下一条线段。如果 S_i 在无限带 $|x-a|\leqslant\mu$ 之外，那么它不能与圆相交。因此，排除测试为

$$|x_i-a|\leqslant\mu \text{ 且 } |x_{i+1}-a|\leqslant\mu \text{ 且 } (x_i-a)(x_{i+1}-a)>0$$

最开始的两个条件保证每一条线段的端点在带外。最后一个条件保证端点都在带的同一边。类似地，如果 S_i 在无限带 $|y-b|\leqslant\mu$ 之外，那么它不能与圆相交，排除测试为

$$|y_i-b|\leqslant\mu \text{ 且 } |y_{i+1}-b|\leqslant\mu \text{ 且 } (y_i-b)(y_{i+1}-b)>0$$

图 6.4 说明了上面的论述。线段 S_1 虽然在圆外，但并未被排除，因为它部分地位于每一无限带内。然而，S_2 被排除，因为它在垂直带之外。

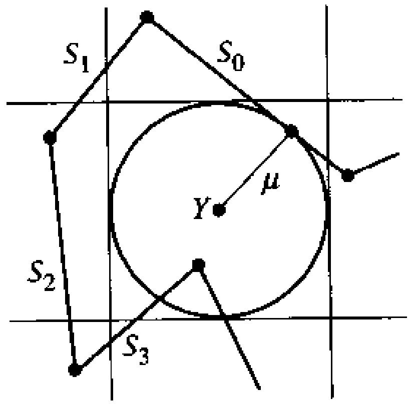


图 6.4　从线段 S_0 中得到折线和 Y 之间的最小距离 μ。S_1 和 S_2 不能改变 μ，因为它们位于圆心为 Y、半径为 μ 的圆之外。线段 S_3 不会改变 μ，因为它与该圆相交。无穷带测试不会排除 S_1 和 S_3，因为它们部分地同时位于两个无穷带中；然而，S_2 被排除，因为它位于垂直带之外。矩形测试排除 S_1 和 S_2，因为它们位于包含该圆的矩形之外；然而，S_3 不会被排除

由于在中间计算中应该避免平方根计算，因此实现算法保存 μ^2，而不保存 μ。排除测试必须基于 μ^2 重新建立：

$$|x_i-a|^2\leqslant\mu^2 \text{ 且 } |x_{i+1}-a|^2\leqslant\mu^2 \text{ 且 } (x_i-a)(x_{i+1}-a)>0$$

或者

$$|y_i-b|^2\leqslant\mu^2 \text{ 且 } |y_{i+1}-b|^2\leqslant\mu^2 \text{ 且 } (y_i-b)(y_{i+1}-b)>0$$

在排除测试中用到的数值同时也用于距离平方的计算，因此这些值可以临时保存起来，以备后面使用，从而避免重复计算。此外，当前测试中的数值 $x_{i+1}-a$ 和 $y_{i+1}-b$ 在下次测试中就是 x_i-a 和 y_i-b，因此这些值也可以临时保存起来，以备后面使用，以避免重复计算。

排除测试的一种改进方法是利用线段和包含以 Y 为圆心、μ 为半径的圆的轴对齐矩形之间的相交测试。我们可以利用 7.7 节讨论的轴分解方法。图 6.4 说明了这种改进。在前

一种方法中，线段S_1未被排除，因为它部分地位于每一个无限带内。然而，S_1被当前的方法排除，因为它与轴对齐矩形不相交。

如果折线的线段采用对称形式$C+t\hat{u}$，其中C为线段的中点，$\hat{u}$为单位长度向量，且$|t|\leqslant r$，那么，排除测试如下。定义$\vec{\Delta}=C-Y=(\Delta_0,\Delta_1)$且$\hat{u}=(u_0,u_1)$。如果如下的任何测试为真，那么线段被排除：

$$
\begin{aligned}
&|\Delta_0|\geqslant \mu+r|u_0|\\
&|\Delta_1|\geqslant \mu+r|u_1|\\
&|\Delta_0u_1-\Delta_1u_0|\geqslant r\mu(|u_0|+|u_1|)
\end{aligned}
$$

由于要避免平方根计算而采用保持μ^2的值，因此上述三个测试必须做一点修改：

$$|\Delta_0|-r|u_0|\geqslant 0\text{且}(|\Delta_0|-r|u_0|)^2\geqslant\mu^2$$

$$|\Delta_1|-r|u_1|\geqslant 0\text{且}(|\Delta_1|-r|u_1|)^2\geqslant\mu^2$$

$$|\Delta_0u_1-\Delta_1u_0|^2\geqslant r^2\mu^2(|u_0|+|u_1|)^2$$

最后，如果应用程序具有特定的关于其折线的形式的信息，那么可能可以建立一种存储折线的数据结构，这种数据结构有助于局部计算。当然，这种算法只能用于特定的应用，而不适用于通用的工具。

6.3 点到多边形的距离

测量点与多边形之间的距离与测量点与折线之间的距离的惟一区别就是，多边形被当做实心对象来处理。如果点在多边形内，那么距离为零。如果点在多边形外，那么，对于非凸的简单多边形，可以不做任何修改地使用点与折线的距离算法。参见13.3节中关于点与多边形的包含测试。

我们在此处理几种特殊情形。我们考虑如下的特殊情形，即计算点与三角形、矩形和垂直平截头体之间的距离。这些多边形都是凸多边形。我们也提及一些计算点与凸多边形之间距离的算法。

6.3.1 点到三角形的距离

设Y为测试点，并设三角形具有逆时针次序的顶点P（$0\leqslant i\leqslant 2$）。如果Y在实心三角形内，则距离定义为零。如果Y在实心三角形之外，那么问题简化为寻找（作为折线的）三角形上与测试点最接近的点。算法限定于通过确定Y在何处临近三角形来搜索最接近点。与直接计算测试点与三条边的距离并选取极小值相比，这样做效率更高。图6.5说明了实际情形。图6.5（a）显示了与三角形距离为零的一个点，因为该点在三角形之内；图6.5（b）显示了一个最接近三角形一条边的点；图6.5（c）显示了一个最接近三角形一个顶点的点。实线表示包含三角形边的直线。点线表示边的垂直方向。图6.5（d）显示了一个最接近三角形一条边的点，但是它位于两条边射线形成的原点为一个顶点的楔形区域内。这个例子说明，三角形上与Y最接近的点取决于Y在由顶点上与边垂直的直线所分区的平面上的位置，而不是取决于边所在的直线本身。特别地，其中的差别显示在三角形的钝角上。

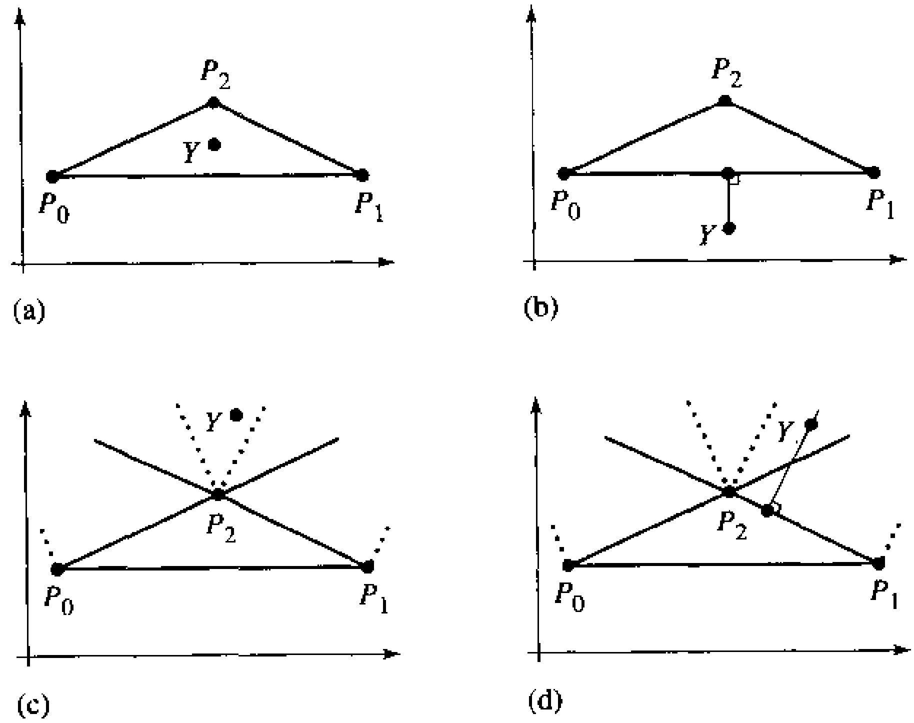


图 6.5　三角形上距给定点最近的点：（a）$\text{Dist}(Y,\mathcal{T})=0$；（b）$\text{Dist}(Y,\mathcal{T})=\text{Dist}\ (Y,<P_0,P_1>)$；（c）$\text{Dist}(Y,\mathcal{T})=\text{Dist}(Y,P_2)$；（d）$\text{Dist}(Y,\mathcal{T})=\text{Dist}(Y,<P_1,P_2>)$

我们提供两种搜索最接近点的方法。第一种方法首先有效地搜索三角形内部，然后再搜索其边。其主要目标是最多允许进行一次除法，并且仅在绝对必要时才进行除法。为了避免费时的除法运算，代之以进行更多的浮点比较。在现在的计算机体系结构中，浮点加法和乘法运算比进行浮点比较的速度更快，如果应用程序需要进行大量的点与三角形之间的距离计算，这有可能是一个问题。第二种方法有效地进行次序相反的搜索，即先搜索边，再搜索其内部。其主要目标是希望最接近点是一个顶点，如果如此，最接近点就可以很快地被找到。

1. 从内点到边的最接近点搜索

该算法使用三角形的参数形式。设 $\vec{d}_0=P_1-P_0$ 且 $\vec{d}_1=P_2-P_0$。三角形上的点为 $X(t_0,t_1)=P_0+t_0\vec{d}_0+t_1\vec{d}_1$，其中 $t_0\geqslant 0$，$t_1\geqslant 0$ 且 $t_0+t_1\leqslant 1$。点 Y 与三角形上的点 $X(t_0,t_1)$ 的距离平方可以表示为如下的二次方程

$$F(t_0,t_1)=\|X(t_0,t_1)-Y\|^2=\|P_0+t_0\vec{d}_0+t_1\vec{d}_1-Y\|^2$$

$$=a_{00}t_0^2+2a_{01}t_0t_1+a_{11}t_1^2-2b_0t_0-2b_1t_1+c$$

其中 $a_{00}=\|\vec{d}_0\|^2$，$a_{01}=\vec{d}_0\cdot\vec{d}_1$，$a_{11}=\|\vec{d}_1\|^2$，$b_0=\vec{d}_0\cdot(Y-P_0)$，$b_1=\vec{d}_1\cdot(Y-P_0)$ 并且 $c=\|Y-P_0\|^2$。虽然参数 t_0 和 t_1 服从上述三角形的限制条件，我们将 $F(t_0,t_1)$ 看成一个关于 t_0 和 t_1 的所有值的函数。所有数对 (t_0,t_1) 的集合叫做参数平面。当满足下式时，函数 F 将取全局最小值：

$$(0,0)=\vec{\nabla}F=2\left(a_{00}t_0+a_{01}t_1-b_0,a_{01}t_0+a_{11}t_1-b_1\right)$$

该方程系统的解为

$$\bar{t}_0=\frac{a_{11}b_0-a_{01}b_1}{a_{00}a_{11}-a_{01}^2} \text{ 且 } \bar{t}_1=\frac{a_{00}b_1-a_{01}b_0}{a_{00}a_{11}-a_{01}^2}$$

与 Y 最接近的三角形点取决于$(\bar{t}_0,\bar{t}_1)$在参数平面上的位置。图 6.6 显示了参数平面被包含三角形边的直线划分为 7 个区域。如果 $(\bar{t}_0,\bar{t}_1)$ 位于区域 0，那么 Y 在三角形内，且距离为零。对于其他的区域，注意 $F(t_0,t_1)=\lambda>0$ 定义的阶层曲线为椭圆（关于阶层曲线的定义和讨论，参见 A.9.1 节）。如果 $(\bar{t}_0,\bar{t}_1)$ 位于区域 1，那么最接近点在边 $t_0+t_1=1$ 上。如果存在值 $\lambda_0>0$ 使得对应的阶层曲线与边相切，那么相交的点 $(\hat{t}_0,\hat{t}_1)$ 就是最接近 Y的点。也可能没有阶层曲线与边相切，此时最接近 Y 的点必然就是边的端点。图 6.7 说明了这两种情形。如果 $(\bar{t}_0,\bar{t}_1)$ 位于区域 3 和 5 时，情形与此类似。图 6.7（a）显示了与边的相切。图 6.7（b）显示了与顶点的接触。

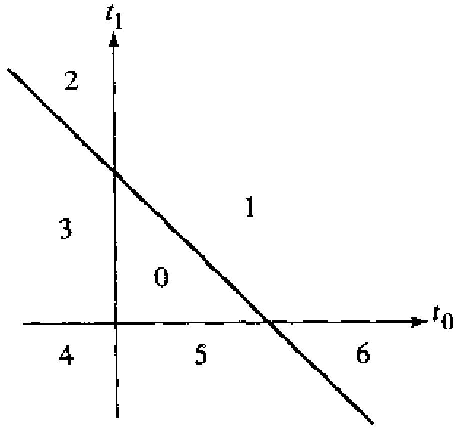


图 6.6　将参数平面划分为 7 个区域

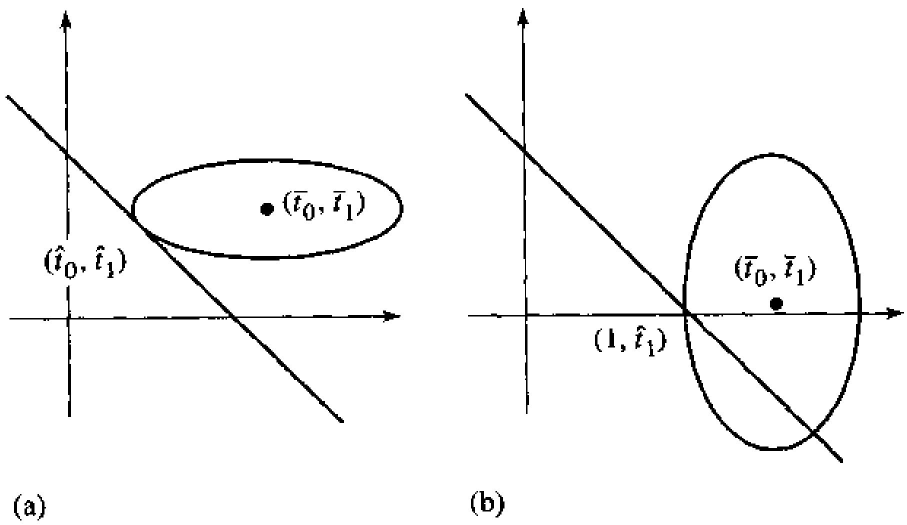


图 6.7　阶层曲线 $F(t_0,t_1)$ 与三角形的接触点：（a）与一条边相切；（b）与一个顶点接触

如果 $(\bar{t}_0,\bar{t}_1)$ 位于区域 2，则存在三种可能。如果存在一个值 $\lambda_0>0$ 使得对应的阶层曲线与 $t_0+t_1=1$所包含的边相切，那么相交的点 $(\hat{t}_0,\hat{t}_1)$ 就是最接近于 Y 的点。如果不存在阶层曲线与这条边相切，那么可能存在一条阶层曲线与 $t_0=0$ 所包含的边相切。相交的点 $(0,\hat{t}_1)$

就是最接近 Y 的点。如果不存在阶层曲线与两条边相切，那么对应于参数对（0，1）的顶点就是最接近 Y 的点。图 6.8 说明了这三种情形。如果 $(\bar{t}_0, \bar{t}_1)$ 位于区域 4 或 6，则情形与此类似。图 6.8（a）显示了与一条边的相切。图 6.8（b）显示了与另一条边的相切接触。图 6.8（c）显示了与一个顶点的接触。

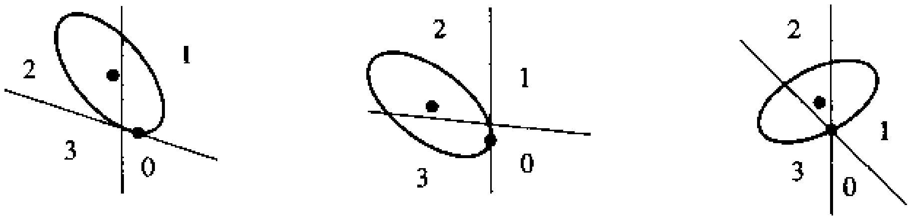


图 6.8 阶层曲线 $F(t_0, t_1)$ 与三角形的接触点：（a）与一条边相切；（b）与另一条边相切；（c）与一个顶点接触

如下的代码段设定最多仅出现一次除法。

```
float SquaredDistance(Point Y, Triangle T)
{
    // coefficients of F(t0, t1), calculation of c is deferred until needed
    Point D0 = T.P1 - T.P0, D1 = T.P2 - T.P0, Delta = Y - T.P0;
    float a00 = Dot(D0, D0), a01 = Dot(D0, D1), a11 = Dot(D1, D1);
    float b0 = Dot(D0, Delta), b1 = Dot(D1, Delta);

    // Grad F(t0, t1) = (0, 0) at (t0, t1) = (n0 / d, n1 / d)
    float n0 = a11 * b0 - a01 * b1;
    float n1 = a00 * b1 - a01 * b0;
    float d  = a00 * a11 - a01 * a01;  // theoretically positive

    if (n0 + n1 <= d) {
        if (n0 >= 0) {
            if (n1 >= 0) {
                region 0
            } else {
                region 5
            }
        } else if (n1 >= 0) {
            region 3
        } else {
            region 4
        }
    } else if (n0 < 0) {
        region 2
    } else if (n1 < 0) {
        region 6
    } else {
       region 1
    }
}
```

由于 Y 在三角形内且距离平方为零，因此处理区域 0 的代码段仅仅返回零。

```
// Region 0.  Point is inside the triangle, squared distance is zero
return 0;
```

如果$(\bar{t}_0, \bar{t}_1)$位于区域 5，那么距离平方函数简化为

$$G(t_0) = F(t_0, 0) = a_{00}t_0^2 - 2b_0t_0 + c$$

现在的问题是通过求$G(t_0)$（$t_0 \in [0, 1]$）的极小值来计算$\hat{t}_0$。事实上，这与求 F 的极小值属于同一类问题，只是少一维而已。极小值出现在$G' = 0$处或者区间的一个端点上。对$G' = 2(a_{00}t_0 - b_0) = 0$的解为$t_0 = b_0/a_{00}$。如果$t_0 \in (0, 1)$，那么$\hat{t}_0 = b_0/a_{00}$。如果$t_0 \leqslant 0$，那么$\hat{t}_0 = 0$。否则，如果$t_0 \geqslant 1$ 则$\hat{t}_0 = 1$。处理区域 5 的代码段为

```
// Region 5.  Minimize G(t0) = F(t0, 0) for t0 in [0, 1].  G'(t0) = 0 at
// t0 = b0 / a00.

float c = Dot(Delta, Delta);
if (b0 > 0) {
    if (b0 < a00) {
        // closest point is interior to the edge
        return c - b0 * b0 / a00;  // F(b0 / a00, 0)
    } else {
        // closest point is end point (t0, t1) = (1, 0)
        return a00 - 2 * b0 + c;  // F(1, 0)
    }
} else {
    // closest point is end point (t0, t1) = (0, 0)
    return c;  // F(0, 0)
}
```

类似的简化出现在处理区域 3 的代码中。代码段为

```
// Region 3.  Minimize G(t1) = F(0, t1) for t1 in [0, 1].  G'(t1) = 0 at
// t1 = b1 / a11.

float c = Dot(Delta, Delta);
if (b1 > 0) {
    if (b1 < a11) {
        // closest point is interior to the edge
        return c - b1 * b1 / a11;  // F(0, b1 / a11)
    } else {
        // closest point is end point (t0, t1) = (0, 1)
        return a11 - 2 * b1 + c;  // F(0, 1)
    }
} else {
    // closest point is end point (t0, t1) = (0, 0)
    return c;  // F(0, 0)
}
```

类似的简化也出现在处理区域 1 的代码中，只是代数运算更复杂一些。需要求极小值的函数为

$$G(t_0) = F(t_0, 1 - t_0) = (a_{00} - 2a_{01} + a_{11})t_0^2 + 2(a_{01} - a_{11} - b_0 + b_1)t_0 + (a_{11} - 2b_1 + c)$$

对$G'=0$的解为$t_0=(a_{11}-a_{01}+b_0-b_1)/(a_{00}-2a_{01}+a_{11})$。理论上，分母为正。

```
// Region 1. Minimize G(t0) = F(t0, 1 - t0) for t0 in [0, 1]. G'(t0) = 0 at
// t0 = (a11 - a01 + b0 - b1) / (a00 - 2 * a01 + a11).

float c = Dot(Delta, Delta);
float n = a11 - a01 + b0 - b1, d = a00 - 2 * a01 + a11;
if (n > 0) {
    if (n < d) {
        // closest point is interior to the edge
        return (a11 - 2 * b1 + c) - n * n / d;  // F(n / d, 1 - n / d)
    } else {
        // closest point is end point (t0, t1) = (1, 0)
        return a00 - 2 * b0 + c;  // F(1, 0)
    }
} else {
    // closest point is end point (t0, t1) = (0, 1)
    return a11 - 2 * b1 + c;  // F(0, 1)
}
```

正如我们前面说过的，对区域 2 的分析更复杂，因为最接近的点可能出现在两条边的任何一条上。伪码采用的测试是判断最接近的点是否在边$t_0=0$上。如果是，则计算距离，函数返回。如果不是，则最接近的点在另一条边$t_0+t_1=1$上，计算距离，函数返回。

```
// Region 2.  Minimize G(t1) = F(0, t1) for t1 in [0, 1].  If t1 < 1, the
// parameter pair (0, max{0, t1}) produces the closest point.  If t1 = 1,
// then minimize H(t0) = F(t0, 1 - t0) for t0 in [0, 1].  G'(t1) = 0 at
// t1 = b1 / a11.  H'(t0) = 0 at t0 = (a11 - a01 + b0 - b1) / (a00 - 2 * a01
// + a11).

float c = Dot(Delta, Delta);

// minimize on edge t0 = 0
if (b1 > 0) {
    if (b1 < a11) {
        // closest point is interior to the edge
        return c - b1 * b1 / a11;  // F(0, b1 / a11)
    } else {
        // minimize on the edge t0 + t1 = 1
        float n = a11 - a01 + b0 - b1 d = a00 - 2 * a01 + a11;
        if (n > 0) {
            if (n < d) {
                // closest point is interior to the edge
                return (a11 - 2 * b1 + c) - n * n / d;  // F(n / d, 1 - n / d
            } else {
                // closest point is end point (t0, t1) = (1, 0)
                return a00 - 2 * b0 + c;  // F(1, 0)
            }
        } else {
```

```
                // closest point is end point (t0, t1) = (0, 1)
                return a11 - 2 * b1 + c;  // F(0, 1)
            }
        }
    } else {
        // closest point is end point (t0, t1) = (0, 0)
        return c;  // F(0, 0)
    }
```

处理区域 6 的伪码的实现与上述方法相似：

```
// Region 6.  Minimize G(t0) = F(t0, 0) for t0 in [0, 1].  If t0 < 1, the
// parameter pair (max{0, t0}, 0) produces the closest point.  If t0 = 1,
// then minimize H(t1) = F(t1, 1 - t1) for t1 in [0, 1].  G'(t0) = 0 at
// t0 = b0 / a00.  H'(t1) = 0 at t1 = (a11 - a01 + b0 - b1) / (a00 - 2 * a01
// + a11).

float c = Dot(Delta, Delta);

// minimize on edge t1 = 0
if (b0 > 0) {
    if (b0 < a00) {

        // closest point is interior to the edge
        return c - b0 * b0 / a00;  // F(b0 / a00, 0)
    } else {
        // minimize on the edge t0 + t1 = 1
        float n = a11 - a01 + b0 - b1, d = a00 - 2 * a01 + a11;
        if (n > 0) {
            if (n < d) {
                // closest point is interior to the edge
                return (a11 - 2 * b1 + c) - n * n / d;  // F(n / d, 1 - n / d)
            } else {
                // closest point is end point (t0, t1) = (1, 0)
                return a00 - 2 * b0 + c;  // F(1, 0)
            }
        } else
            // closest point is end point (t0, t1) = (0, 1)
            return a11 - 2 * b1 + c;  // F(0, 1)
        }
    }
} else {
    // closest point is end point (t0, t1) = (0, 0)
    return c;  // F(0, 0)
}
```

最后，处理区域 4 的伪码为：

```
// Region 4.  Minimize G(t0) = F(t0, 0) for t0 in [0, 1].  If t0 > 1, the
// parameter pair (min{1, t0}, 0) produces the closest point.  If t0 = 0,
// then minimize H(t1) = F(0, t1) for t1 in [0, 1].  G'(t0) = 0 at
// t0 = b0 / a00.  H'(t1) = 0 at t1 = b1 / a11.
```

```
float c = Dot(Delta, Delta);

// minimize on edge t1 = 0
if (b0 < a00) {
    if (b0 > 0) {
        // closest point is interior to edge
        return c - b0 * b0 / a00;  // F(b0 / a00, 0)
    } else {
        // minimize on edge t0 = 0
        if (b1 < a11) {
            if (b1 > 0) {
                // closest point is interior to edge
                return c - b1 * b1 / a11;  // F(0, b1 / a11)
            } else {
                // closest point is end point (t0, t1) = (0, 0)
                return c;  // F(0, 0)
            }
        } else {
            // closest point is end point (t0, t1) = (0, 1)
            return a11 - 2 * b1 + c;  // F(0, 1)
        }
    }
} else {
    // closest point is end point (t0, t1) = (1, 0)
    return a00 - 2 * b0 + c;  // F(1, 0)
}
```

2. 搜索从内点到边的最接近点的时间分析

这里列出了上述伪码的运算统计，以提供该代码的最佳和最差表现。我们统计加法 A，乘法 M，除法 D，两个非零浮点数的比较C_T，以及其中一个为零的两个浮点数的比较C_Z。我们将比较进行这样的划分，是因为浮点函数库倾向于支持数值的符号位测试，这样比一般的浮点数比较更快一些。

代码块SquaredDistance中出现在一组条件语句之前，但包含第一个条件内的加法的代码段，要求进行 15 次加法和 16 次乘法。处理每一个区域的代码段都要进行这些运算。表 6.1 显示了处理不同区域的最佳和最差情形的运算统计。由于设计的原因，我们可以预期，处理区域 0 的最佳情形处理每一个点的时间最短。处理区域 6 的最差情形处理每一个点的时间最长。

表 6.1 使用内点到边的方法计算点到三角形距离的运算结果统计

区域 / 统计	A	M	D	C_T	C_Z
0 / 最佳	15	16	0	1	2
0 / 最差	15	16	0	1	2
1 / 最佳	23	20	0	1	3
1 / 最差	24	21	1	2	3
2 / 最佳	16	18	0	1	2
2 / 最差	24	21	1	3	3
3 / 最佳	16	18	0	1	3

续表

区域 / 统计	A	M	D	C_T	C_Z
3 / 最差	17	19	1	2	3
4 / 最佳	18	19	0	2	2
4 / 最差	17	19	1	3	4
5 / 最佳	16	18	0	1	3
5 / 最差	17	19	1	2	3
6 / 最佳	16	18	0	1	3
6 / 最差	24	21	1	3	4

3. 从边到内点的最接近点搜索

Gino van den Bergen 在一篇登载在新闻组 *comp.graphics.algorithms* 上的帖子中提出了这种方法，试图通过先计算到边的距离，并期望两条边的公共顶点是最接近的点，来提高计算速度。该帖子中的意见是，当 Y 离三角形很远时，这种方法显然应该比前面讨论的方法表现更好。这种方法的基础是，如果为三角形选择一个大范围的区域，并且测试点非均匀地分布在该区域内，那么一个顶点更接近于测试点的可能性比一条边上的点更接近于测试点或测试点在三角形内的可能性要大得多。为了证明这一点，考虑一个顶点为（0，0），（1，0）和（0，1）的三角形，范围为$[-r,r]^2$，其中$r \geq 1$。图 6.9 说明了这种情形，并显示了最接近于顶点和边的点的区域。

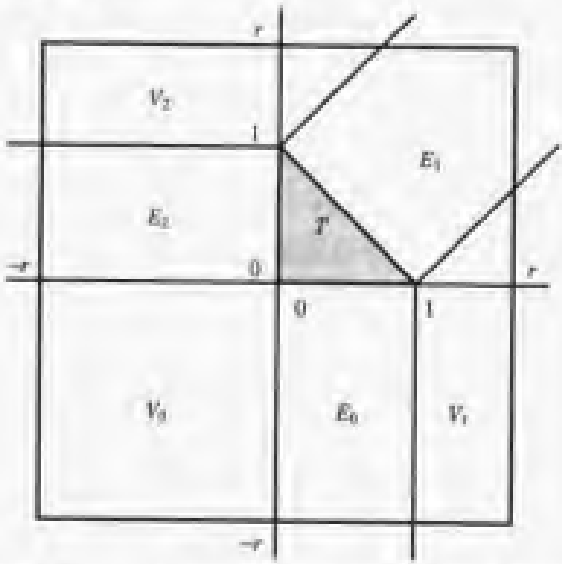


图 6.9　一个三角形，一个矩形有界框和距它们的顶点和边最近的点域

区域 V_0，V_1和V_2分别是最接近于顶点（0，0），（1，0）和（0，1）的点集。区域E_0，E_1和E_2分别是最接近于边$\langle(0,0),(1,0)\rangle$，$\langle(1,0),(0,1)\rangle$和$\langle(0,1),(0,0)\rangle$的点集。区域 T 是三角形的内部区域。T 的面积为$A_T = 1/2$。边区域的总面积为 $A_E = 4r - 3/2$。顶点区域的总面积为 $A_V = 4r^2 - 4r + 1$。显然，对于足够大的 r，$A_V > A_E$，因为A_V 是 r 的二次函数，而 A_E 只是 r 的线性函数。因此，对于足够大的 r，在矩形内随机选择点，点位于顶点区域的可能性最大。因此，在这种情形下，在测试与测试点的接近性时，使用先测试顶点的算法是合理的。

然而，我们现在来考虑小的 r。如果 $r=1$，则惟一的面积为正的顶点区域是V_0，其面积为$A_V = 1$。边区域的面积为$A_E = 5/2 > A_V$。一般地，对于$1 \leqslant r \leqslant 1+\sqrt{6}/4 \doteq 1.612$，$A_E \geqslant A_V$。对于这一范围的 r，在矩形内随机选择一个点，该点位于边区域的可能性最大。这说明，当应用程序中测试点实际上是分布于如上所述的较小的范围时，最好根据你自己的数据来选择测量点与三角形之间的距离的方法。

下面列出了刚才讨论的算法的伪码。代码返回了与测试点最接近的三角形上的点。测试点与三角形的距离可通过这种计算而得到。

```
float SquaredDistance (Point Y, Triangle T)
{
    // triangle vertices V0, V1, V2,  edges E0=<V0, V1>,  E1=<V1, V2>, E2=<V2, V0>

    // closest point on E0 to P is K0 = V0 + t0 * (V1 - V0) for some t0 in [0, 1]
    float t0 = ParameterOfClosestEdgePoint(P, E0);

    // closest point on E1 to P is K1 = V1 + t1 * (V2 - V1) for some t1 in [0, 1]
    float t1 = ParameterOfClosestEdgePoint(P, E1);

    if (t0 == 0 and t1 == 0)  // closest point is vertex V1
        return SquaredLength(Y - V1);

    // closest point on E2 to P is K2 = V2 + t2 * (V0 - V2) for some t2 in [0, 1]
    float t2 = ParameterOfClosestEdgePoint(P, E2);

    if (t1 == 0 and t2 == 0)  // closest point is vertex V2
        return SquaredLength(Y - V2);
    if (t0 == 0 and t2 == 0)  // closest point is vertex V0
        return SquaredLength(Y - V0);

    // Y = c0 * V0 + c1 * V1 + c2 * V2 for c0 + c1 + c2 = 1
    GetBarycentricCoordinates(Y, V0, V1, V2, c0, c1, c2);

    if (c0 < 0)  // closest point is K1 on edge E1
        return SquaredLength(Y - (V1 + t1 * (V2 - V1)));

    if (c1 < 0)  // closest point is K2 on edge E2
        return SquaredLength(Y - (V2 + t2 * (V0 - V2)));

    if (c2 < 0)  // closest point is K0 on edge E0
        return SquaredLength(Y - (V0 + t0 * (V1 - V0)));

    return 0;  // Y is inside triangle
}
```

函数 ParameterOfClosestEdgePoint(P,E)其实就是用于计算点与线段之间距离的那个函数。P 在包含边(V_0, V_1)的直线上的投影是$K = V_0 + t(V_1 - V_0)$，其中$t = (P - V_0)\ /\|V_1 - V_0\|^2$。如果$t < 0$，那么它将收缩于$t = 0$，最接近于 P 的点为V_0。如果$t > 1$，那么它将收缩于$t = 1$，最接近于 P 的点为V_1。否则，$t \in [0,1]$，最接近的点是 K。上述的函数返回 t 的值。如果函

数按上述的方法实现，那么其中包括一个与边的长度的平方相关的除法运算。该函数至少要被调用两次，因此，距离计算将至少包含两次除法运算，这要花费很长的时间。一种更聪明的实现方法不需要进行除法运算，但是要计算 t 的分子 n 和分母 d。如果$n < 0$，则 t 收缩于 0。如果$n > d$，则 t 收缩于 1。分子和分母的值应该存储在函数体的局部变量中，以供以后计算重心坐标时使用。如果函数返回三个顶点中的一个，那么对所有 t 值的除法 n/d都不会进行。

函数GetBarycentricCoordinates计算$P = \sum_{i=0}^{2} c_i V_i$，其中$\sum_{i=0}^{2} c_i = 1$。一旦已知$c_1$和$c_2$，我们就可求得$c_0 = 1 - c_1 - c_2$。关于 P 的方程等价于$P - V_0 = c_1(V_1 - V_0) + c_2(V_2 - V_0)$。该向量方程表示两个带两个未知数$c_1$和$c_2$的线性方程，该系统可以用普通的方法来求解。如果用直接的方法来实现，则求解过程需要一次与系数矩阵的行列式相关的除法。该除法运算是不必执行的。重心计算函数可以用三个具有相同分母的有理数 $c_i = n_i/d$的形式来返回坐标。分子和分母以不同的存储值返回。符号测试$c_0 < 0$等价于 $n_0 d < 0$，因此除法被一次乘法所代替。其中的条件测试是一次符号测试，因此通常的浮点数比较可用浮点数的符号位测试（一般快一些）来代替。甚至更好的情形是，可以避免乘法 $n_i d$，仅仅需要一条测试 d 的符号位的条件测试语句。每一条测试语句都有三个测试 n_i的符号位的条件测试。

如果 $c_0 < 0$，那么最接近的点在边 E_1和$K_1 = V_1 + t_1(V_2 - V_1)$上。$t_1$的确定值是必需的。如果调用ParameterOfClosestEdgePoint时推迟了除法运算n_1/d_1，那么现在必须进行该除法运算，以计算出 K_1。相似的方法也适用于对c_1 和c_2 的条件语句。

使用推迟除法并且在重心计算函数中避免除法的更详细的伪码如下所示。其中标出了返回语句，以用于下一节进行伪码的时间分析。

```
float SquaredDistance (Point Y, Triangle T)
{
    // T has vertices V0, V1, V2

    // t0 = n0/d0 = Dot(Y - V0, V1 - V0) / Dot(V1 - V0, V1 - V0)
    Point D0 = Y - V0, E0 = V1 - V0;
    float n0 = Dot(D0, E0);

    // t1 = n1/d1 = Dot(Y - V1, V2 - V1) / Dot(V2 - V1, V2 - V1)
    Point D1 = Y - V1,  E1 = V2 - V1;
    float n1 = Dot(D1, E1);

    if (n0 <= 0 and n1 <= 0)  // closest point is V1
        return Dot(D1, D1);  // RETURN 0

    // t2 = n2/d2 = Dot(Y - V2, V0 - V2) / Dot(V0 - V2, V0 - V2);
    Point D2 = Y - V2,  E2 = V0 - V2;
    float n2 = Dot(D2, E2);

    if (n1 <= 0 and n2 == 0)  // closest point is V2
        return Dot(D2, D2);  // RETURN 1

    if (n0 <= 0 and n2 <= 0)  // closest point is V0
        return Dot(D0, D0);  // RETURN 2
```

```
// D0 = Y - V0 = V0 + c1 * (V1 - V0) + c2 * (V2 - V0) = V0 + c1
// * E1 - c2 * E2 for
// c0 + c1 + c2 = 1, c0 = m0 / d, c1 = m1 / d, c2 = m2 / d
float e00 = Dot(E0, E0), e02 = Dot(E0, E2), e22 = Dot(E2, E2);
float d = e02 * e02 - e00 * e22;
float a = Dot(D0, E2);
float m1 = e02 * a - e22 * n0;
float m0, m2;
Point D;
if (d > 0) {
    if (m1 < 0) {  // closest point is V2 + t2 * E2
        t2 = n2 / e22;
        D = Y - (V2 + t2 * E2);
        return Dot(D, D);  // RETURN 3a
    }

    m2 = e00 * a - e02 * n0;
    if (m2 < 0) { // closest point is V0 + t0 * E0
        t0 = n0 / e00;
        D = Y - (V0 + t0 * E0);
        return Dot(D, D);  // RETURN 4a
    }

    m0 = d - m1 - m2;
    if (m0 < 0) { // closest point is V1 + t1 * E1
        t1 = n1/Dot(E1, E1);
        D = Y - (V1 + t1 * E1);
        return Dot(D, D);  // RETURN 5a
    }
} else {
    if (m1 > 0) {  // closest point is V2 + t2 * E2
        t2 = n2 / e22;
        D = Y - (V2 + t2 * E2);
        return Dot(D, D);  // RETURN 3b
    }

    m2 = e00 * a - e02 * n0;
    if (m2 > 0) { // closest point is V0 + t0 * E0
        t0 = n0 / e00;
        D = Y - (V0 + t0 * E0);
        return Dot(D, D);  // RETURN 4b
    }

    m0 = d - m1 - m2;
    if (m0 > 0) { // closest point is V1 + t1 * E1
        t1 = n1 / Dot(E1, E1);
        D = Y - (V1 + t1 * E1);
        return Dot(D, D);  // RETURN 5b
    }
```

```
        }

        return 0;  // Y is inside triangle, RETURN 6
    }
```

4. 搜索从边到内点的最接近点的时间分析

这里列出了上述伪码的运算统计，以提供该代码的最佳和最差表现。我们统计加法 A，乘法 M，除法 D，以及其中一个为零的两个浮点数的比较 C_Z。该伪码中没有出现一般的浮点数比较，因此在前一个算法中定义的C_T总是为零。表 6.2 显示了伪码中每一个返回语句的运算统计。最差的情形被假定为最开始的三个条件对的三个返回块，即如果第二个条件测试为假，则需要进行两次符号测试，使得返回语句被跳过。最好的情形是，因为每一个条件的第一个符号测试为假，而使该条件为假，返回被跳过。当函数终止于标记为RETURN 0的返回语句时，出现最好的情形。当函数终止于标记为 RETURN 5a 和RETURN 5b的返回语句时，出现最差的情形。

表 6.2　使用边到内点方法计算点到三角形距离的运算结果统计

返回 / 统计	A	M	D	C_Z
0	11	6	0	2
1	17	10	0	4
2	18	12	0	6
3a，3b	29	28	1	8
4a，4b	30	30	1	9
5a，5b	33	32	1	10
6	27	26	0	10

将该结果与运行结果列于表 6.1 中的另一个算法进行比较，我们将看到边到内点算法的最佳情形$(11A, 6M, 0D, 2C_Z)$比内点到边算法的最佳情形 $(15A, 16M, 0D, 1C_T, 2C_Z)$ 要快。但是，边到内点算法的最差情形$(33A, 32M, 1D, 10C_Z)$比内点到边算法的最差情形$(24A, 21M, 1D, 3C_T, 4C_Z)$要慢。为了决定哪一个算法最适合于你的应用程序，需要进行某些类型的分期数据分析或者计算执行时间的实际经验。

6.3.2　点到矩形的距离

计算点与矩形之间的距离比计算点与三角形之间的距离要简单一些。这种多边形的所有角都是直角，这就极大地简化了问题。在坐标轴与矩形边对齐的坐标系中，这一问题分解为每一维的距离计算问题。

设测试点为 Y。矩形的对称形式为 $X(t_0, t_1) = C + t_0\hat{u}_0 + t_1\hat{u}_1$，其中$|t_0| \leqslant e_0$且$|t_1| \leqslant e_1$。向量$\hat{u}_i$为单位向量，$C$为矩形的中心。这种形式可用来避免任何的除法运算。测试点可变换为$Y = C + s_0\hat{u}_0 + s_1\hat{u}_1$。设$\vec{\Delta} = Y - C$，我们有$s_0 = \hat{u}_0 \cdot \vec{\Delta}$，$s_1 = \hat{u}_1 \cdot \vec{\Delta}$。矩形上与 Y 最接近的点取决于参数平面(t_0, t_1)上 9 个区域的哪一个包含(s_0, s_1)。图 6.10 说明了这些区域。如果(s_0, s_1)在区域 ZZ 内，那么 Y 在矩形内部，距离为零。如果(s_0, s_1)在区域 PZ，ZP，MZ 或 ZM 内，那么，最接近的点是其在矩形对应的边上的投影。否则，(s_0, s_1)在区域 PP，PM，

MP 或 *MM* 内。最接近的点是矩形对应的顶点。

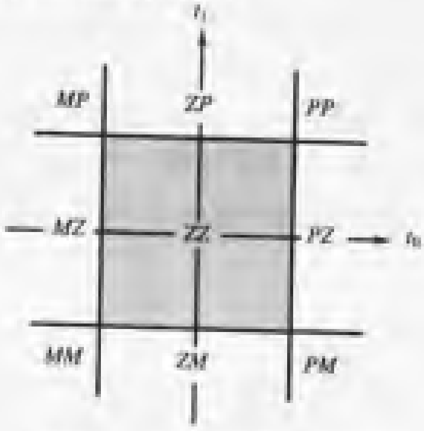


图 6.10 一个矩形对平面的分区

如下伪码的主要代码由嵌套的条件语句所组成，每一条语句对应于参数平面 9 个分区中的一个。然而，这并不是必需的，因为矩形的边是互相垂直的。代码的基本设计思路是分别地处理每一维。

```
float SquaredDistance(Point Y, Rectangle R)
{
    Point Delta = Y - R.C;
    float s0 = Dot(R.U0, Delta), s1 = Dot(R.U1, Delta), sqrDist = 0;

    float s0pe0 = s0 + R.e0;
    if (s0pe0 < 0) {
        sqrDist += s0pe0 * s0pe0;

    } else {
        float s0me0 = s0 - R.e0;
        if (s0me0 > 0)
            sqrDist += s0me0 * s0me0;
    }

    float s1pe1 = s1 + R.e1;
    if (s1pe1 < 0) {
        sqrDist += s1pe1 * s1pe1;
    } else {
        float s1me1 = s1 - R.e1;
        if (s1me1 > 0)
            sqrDist += s1me1 * s1me1;
    }

    return sqrDist;
}
```

6.3.3 点到正交平截面的距离

单锥体（single cone）被定义为一组边界包含两条具有共同原点的射线的点集，射线

的原点称为锥体的顶点。设顶点记为 V，射线具有单位长度方向 $\hat{d}_0$ 和 $\hat{d}_1$。锥体的轴是两条射线的二等分线。设 $\hat{a}$ 为轴的单位长度方向。锥体的角为 $\hat{a}$ 与任一射线方向向量之间的夹角 $\theta \in (0, \pi)$。在本节中，我们仅讨论 $\theta < \pi/2$ 的情形。图 6.11（a）显示了一个单锥。

如果两条平行直线横切地与锥体相交，那么由锥体和这两条直线所围成的凸四边形叫做锥体的一个平截面。图 6.11（b）显示了这样一个平截面。如果直线与锥体的轴垂直，那么平截面叫做正交平截面。图 6.11（c）显示了一个正交平截面。

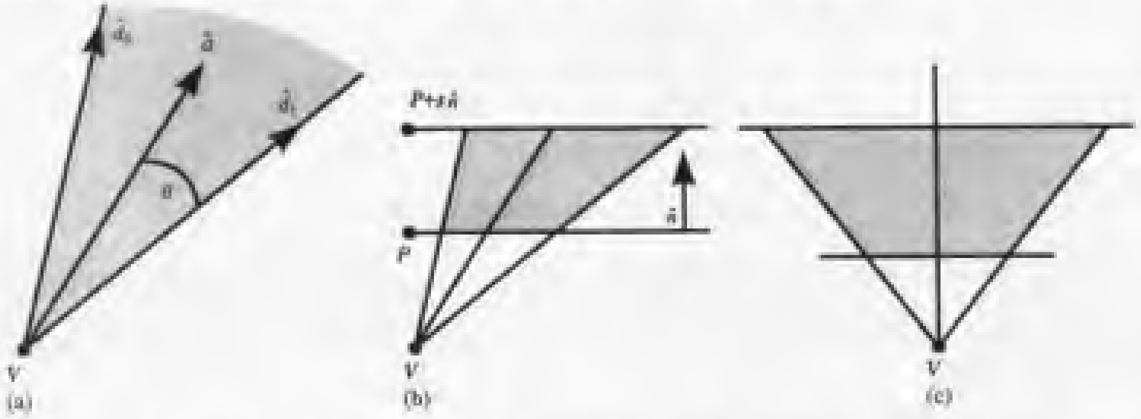


图 6.11　（a）一个单锥的示例；（b）一个锥体的平截面；（c）一个正交平截面

如果 $X - V$ 与 $\hat{a}$ 之间的夹角比 θ 小，则点 X 位于锥体内。我们可用点积 $\hat{a} \cdot (X - V) \geqslant \cos(\theta)$ 来表示这一限制条件。如果最接近于顶点的平行直线包含点 P，并且具有一条指向平截面里面的单位法线 $\hat{n}$，那么其直线方程为 $\hat{n} \cdot (X - P) = 0$。另一条直线包含点 $P + s\hat{n}$（对某些 $s > 0$ 成立）。其直线方程为 $\hat{n} \cdot (X - P) = s$。$X$ 位于平截面内的额外条件为 $0 \leqslant \hat{n} \cdot (X - P) \leqslant s$。如果平面是正交的，则 $\hat{n} = \hat{a}$。

当相机模式为基于透视投影模式时，一个二维空间中的正交平截面是用于三维空间的平截面视图的一种模拟。在二维空间中，V 相当于观察点，两条平行直线相当于近平面和远平面，而两条包围射线相当于平截面视图的左、右扩展。本节提供了一个计算点与正交平截面之间距离的算法。其思想可用于处理三维空间中的相同问题。点与平截面之间距离的计算对可见性测试，特别是当点代表可被画出的网格的包围球体中的点时的可见性测试，是非常有用的。如果包围球体在平截面之外，那么网格是被剔除的，并不需要送往着色程序进行绘制。只要区域的中心与平截面之间的距离大于区域的半径，包围球体就位于平截面之外。注意如下的事实，在三维空间中，如果世界建立在 xy 平面中，而且如果相机的移动被限制为在 xy 平面上的平移和仅相对于向上的向量的旋转，那么包围球体相对左或右平截面平面的可见性测试，可以通过其在 xy 平面上的投影而在二维空间内进行。问题简化为测试一个圆是否在二维正交平截面之外。

计算点与正交平截面之间距离的算法的基础是确定平截面的边和顶点的 Voronoi 区域。包含点的区域可以计算出来。区域内平截面上最接近的点也可以计算出来。距离可以根据它们来计算得到。这种方法可直接从二维空间推广到三维空间，在本书后面的章节中将讨论这一点。

正交平截面有原点 E，单位长度方向向量 $\hat{d}$，垂直单位长度向量 $\hat{l}$。近线具有法线 $\hat{d}$ 并

且包含点$E+n\hat{d}$（对某些$n>0$）。远线具有法线$\hat{d}$并且包含点$E+f\hat{d}$（对某些$f>n$）。平截面的 4 个顶点为$E+n\hat{d}\pm\ell\hat{l}$（对某些$\ell>0$）和$E+(f/n)(n\hat{d}\pm\ell\hat{l})$。设$P$为要计算与平截面的距离的点。该点可在平截面的坐标系中描述为

$$P=E+x_0\hat{l}+x_1\hat{d}$$

因此$x_0=\hat{l}\cdot(P-E)$且$x_1=\hat{d}\cdot(P-E)$。这已足够证明$x_0\geqslant 0$。因为，如果$x_0<0$，通过改变x_0的符号可以得到其反射，并可以计算出最接近的点，那么，该点的反射就是与原来的形状最接近的点。图 6.12 显示了在第一象限内的平截面部分。

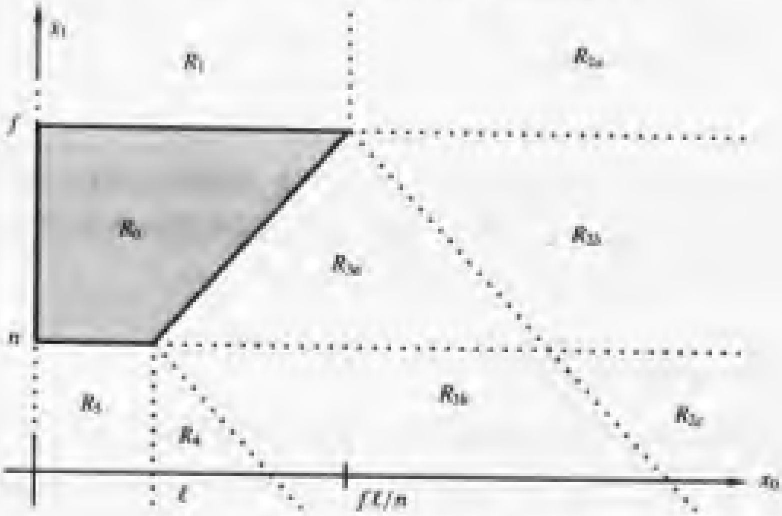


图 6.12 在第一象限内的平截面部分

Voronoi 区域的边界用点线表示。区域R_0包含平截面内的点。区域R_1包含最接近于平截面的顶边的点。区域R_2包含最接近于平截面的顶点$(f\ell/n, f)$的点。根据$\hat{d}$的分量是大于f、在n与f之间还是小于n，可以将该区域划分为三个子区域。区域R_3包含最接近于平截面的斜边的点。根据$\hat{d}$的分量是在n与f之间还是小于n，可以将该区域划分为两个子区域。区域R_4包含最接近于平截面的顶点(ℓ, n)的点。最后，区域R_5包含最接近于平截面的底边的点。

确定(x_0, x_1)的 Voronoi 区域的伪码如下

```
if (x1 >= f) {
    if (x0 <= f * l / n)
        point in R1;
    else
        point in R2a;
} else if (x1 >= n) {
    t = Dot((n, -l), (x0, x1));
    if (t <= 0)
        point in R0;
    else {
        t = Dot((l, n), (x0, x1));
        if (t <= Dot((l, n), (f * l / n,f)))
            point in R3a;
        else
```

```
                point in R2b;
        }
    } else {
        if (x0 <= 1 )
            point in R5;
        else  {
            t = Dot((l, n), (x0, x1));
            if (t <= Dot((l, n), (l, n)))
                point in R4;
            else if (t <= Dot((l, n), (f * l / n, f)))
                point in R3b;
            else
                point in R2c;
        }
    }
```

区域R_1中最接近于点(x_0,x_1)的点是(x_0,f)。区域R_2中最接近的点是$(f\ell/n,f)$。区域R_4中最接近的点是(ℓ,n)。区域R_5中最接近的点是(x_0,n)。区域R_3要求得到分量$(n,-\ell)$关于(x_0,x_1)的投影。最接近的点是$(x_0,x_1)-[(nx_0-\ell x_1)/(\ell^2+n^2)](n,-\ell)$。

6.3.4　点到凸多边形的距离

在搜索点与多边形的边的最小距离时，对于某些特殊的凸多边形来说，并非所有的点与线段的距离都需要进行计算。只有那些对点X可见的边才需要考虑。图 6.13 说明了这一论述。

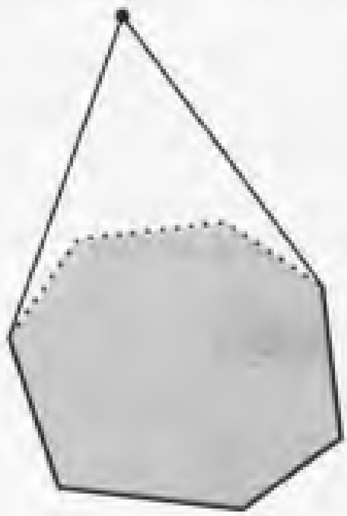


图 6.13　为寻找与测试点最近的点，只需搜索对测试点可见的边。三条可见的边用点线表示。不可见的边用黑线表示。可见的边位于一个以测试点为顶点、边线与该凸多边形相切的锥体内

假定每一条边(P_i,P_{i+1})都有一条相关的指向多边形内部的法线向量$\vec{n}_i$，那么只有当$\vec{n}_i\cdot(X-P_i)\geq 0$时，该边才是可见的。先测试该点积，如果值为负，则计算点与线段之间距离中的除法可被避免。更进一步，可通过检查下一个点与线段之间的距离是否大于当前的点与线段之间的距离再次简化计算。如果与当前边的距离小于或等于与其两条邻边的距离，那么当前的距离就是与多边形边界的距离的最小值。

最后，对于一个给定点和一个凸多边形，6.10 节描述的GJK算法提供了另一种可选的寻找最近点的边界搜索算法。该方法可扩展到更高维的空间，可用于任意的凸形，不论是否多边形或多面体。

6.4 点到二次曲线的距离

一般的二次曲线方程为

$$Q(X) = X^{\mathrm{T}}\mathrm{A}X + B^{\mathrm{T}}X + c = 0$$

其中 **A** 是一个对称的 2×2 矩阵，不一定非是可逆的矩阵；B 是一个 2×1 向量；c 是一个常量。参数 X 是一个 2×1 向量。给定曲线 $Q(X) = 0$ 和一个点 Y，我们需要一个计算曲线上与点 Y 最接近的点的算法。在几何上，最接近的点 X 必须满足 $Y - X$ 是曲线的法线。图 6.14 说明了这一点。由于梯度 $\vec{\nabla}Q(X)$ 是曲线的法线，$Y - X$ 与 $\vec{\nabla}Q(X)$ 必须是平行的，因此，最接近点的代数条件是

$$Y - X = t\vec{\nabla}Q(X) = t(2\mathrm{A}X + B)$$

对某些数量 t 成立。因此

$$X = (\mathrm{I} + 2t\mathrm{A})^{-1}(Y - tB)$$

其中 **I** 为单位矩阵。这个关于 X 的方程可代入一般二次曲线方程，以得到 t 的最高次数为 4 的多项式。

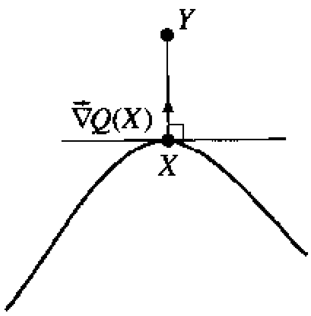


图 6.14 二次曲线上距给定点最近的点

也可以不立即在二次曲线方程中代入 X。我们可以先将其简化为易于实现的形式。将 **A** 进行特征值因式分解，可得 $\mathrm{A} = \mathrm{RDR}^{\mathrm{T}}$，其中 **R** 为标准正交矩阵，其各列为 **A** 的特征向量，其中 $\mathrm{D} = \mathrm{Diag}\{d_0, d_1\}$ 是对角矩阵，其对角线元素为 **A** 的特征值（参见 A.3 节）。因此

$$\begin{aligned}
X &= (\mathrm{I} + 2tA)^{-1}(Y - tB) \\
&= (\mathrm{RR}^{\mathrm{T}} + 2t\mathrm{RDR}^{\mathrm{T}})^{-1}(Y - tB) \\
&= [\mathrm{R}(\mathrm{I} + 2t\mathbf{D})\mathrm{R}^{\mathrm{T}}]^{-1}(Y - tB) \\
&= \mathbf{R}(\mathrm{I} + 2t\mathbf{D})^{-1}\mathbf{R}^{\mathrm{T}}(Y - tB) \\
&= \mathbf{R}(\mathrm{I} + 2t\mathbf{D})^{-1}(\vec{\alpha} - t\vec{\beta})
\end{aligned}$$

其中 $\vec{\alpha} = (\alpha_0, \alpha_1) = \mathrm{R}^{\mathrm{T}}Y$ 且 $\vec{\beta} = (\beta_0, \beta_1) = \mathbf{R}^{\mathrm{T}}B$。在二次曲线方程中替换 X，并简化等式，可得

$$0 = (\vec{\alpha} - t\vec{\beta})^{\mathrm{T}}(\mathrm{I} + 2t\mathrm{D})^{-1}\mathrm{D}(\mathrm{I} + 2t\mathrm{D})^{-1}(\vec{\alpha} - t\vec{\beta}) + \vec{\beta}^{\mathrm{T}}(\mathrm{I} + 2t\mathrm{D})^{-1}(\vec{\alpha} - t\vec{\beta}) + c$$

其逆对角矩阵为$(\mathrm{I} + 2t\mathrm{D})^{-1} = \mathrm{Diag}\{1/(1 + 2td_0), 1/(1 + 2td_1)\}$。用$((1 + 2td_0)(1 + 2td_1))^2$乘以方程两边将得到最高次数为 4 的多项式，$p(t) = p_0 + p_1 t + p_2 t^2 + p_3 t^3 + p_4 t^4$，其中

$$p_0 = c + \alpha_0\beta_0 + \alpha_1\beta_1 + \alpha_0^2 d_0 + \alpha_1^2 d_1$$

$$p_1 = 4[c(d_0 + d_1) + \alpha_0 d_1(\beta_0 + \alpha_0 d_0) + \alpha_1 d_0(\beta_1 + \alpha_1 d_1)] - (\beta_0^2 + \beta_1^2)$$

$$p_2 = 4[c((d_0 + d_1)^2 + 2d_0 d_1) + \alpha_0 d_1^2(\beta_0 + \alpha_0 d_0) + \alpha_1 d_0^2(\beta_1 + \alpha_1 d_1)] - \beta_0^2(4d_1 + d_0) - \beta_1^2(4d_0 + d_1)$$

$$p_3 = 4(d_0 + d_1)[4cd_0 d_1 - (\beta_1^2 d_0 + \beta_0^2 d_1)$$

$$p_4 = 4d_0 d_1[4cd_0 d_1 - (\beta_1^2 d_0 + \beta_0^2 d_1)]$$

可求得$p(t)$的根，并且对每一个根 t 计算$X = (\mathrm{I} + 2tA)^{-1}(Y - tB)$。距离平方的最小值可从对所有根 t 的一组值$\|X(t) - Y\|^2$中选取。

本算法要求能保证数值计算的准确性。如果曲线是一条抛物线，那么$d_0 d_1 = 0$，这时$p_4 = 0$。如果$d_0 d_1$接近于零，那么曲线不是一条抛物线，但是p_4接近于零。多项式的数值根的求解方法必须足够健壮，应该能够处理这种情形。如果曲线是一个圆，且 Y 是圆心，那么圆上的所有点与 Y 的距离都是最小的。该多项式的系数都可确定为零。如果曲线是一个接近于圆的椭圆，那么该多项式的高阶系数可能非常接近于零，这可能导致求根的方法出现问题。

6.5　点到多项式曲线的距离

我们考虑如下曲线$X(t) = \sum_{i=0}^{n} \vec{A}_i t^i$，其中$\vec{A}_n \neq \vec{0}$。设 Y 为测试点。正如二次曲线的情形，最接近的点$X(t)$必须满足条件，即$Y - X(t)$为曲线的法线，但是仅当 t 是函数的定义域的内点时，上述条件才成立。最接近的点可能为曲线的一个端点，与端点的距离可分别计算。对应于最接近的内点，$Y - X(t)$必须垂直于曲线的切线$\vec{X}'(t)$。图 6.15 说明了这一点。内点条件和端点测试是从一个微积分的直接应用中得出的，用以求距离平方函数$F(t) = \|X(t) - Y\|^2$（$t \in I$）的最小值，I 为曲线的定义域区间。当$F'(t) = 0$或位于 I 的一个端点时，F 的全局极小值将出现（如果存在的话）。由于$F(t) = (X(t) - Y) \cdot (X(t) - Y)$，因此其导数的一半为$F'(t)/2 = (X(t) - Y) \cdot \vec{X}'(t)$。右边为两个向量值多项式的点积。其结果是次数为$2n - 1$的数量多项式。计算距离极小值的问题简化为寻找一个多项式的根的问题。

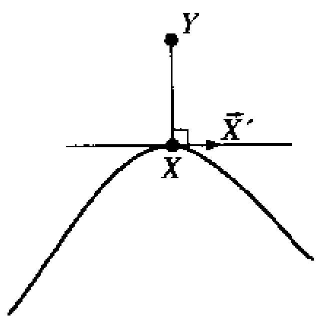


图 6.15　多项式曲线上距给定点最近的点

特别地，定义$\vec{B}_0 = \vec{A}_0 - Y$且$\vec{B}_i = \vec{A}_i$（$i \geqslant 1$）。曲线函数的导数为$\vec{X}'(t) = \sum_{j=0}^{n-1}(j+1)\vec{B}_{j+1}t^j$。定义$\vec{C}_{i,j} = (j+1)\vec{B}_i \cdot \vec{B}_{j+1}$对适当的 i 和 j 成立。那么

$$\begin{aligned} F'(t)/2 &= (X(t) - Y) \cdot \vec{X}'(t) \\ &= \sum_{i=0}^{n}\sum_{j=0}^{n-1}(j+1)\vec{B}_i \cdot \vec{B}_{j+1}t^{i+j} \\ &= \sum_{k=0}^{2n-1}\sum_{m=\max\{0,k-n\}}^{k} C_{k-m,m}t^k \\ &= \sum_{k=0}^{2n-1} D_k t^k \end{aligned} \tag{6.8}$$

其中最后一个等式定义了D_k项。距离的极小值的候选值为满足$F'(t)=0$的那些 t，或者等价地说，这些 t 是$\sum_{k=0}^{2n-1} D_k t^k = 0$的解。

对于较高的次数，求解数值多项式的根的方法可能出现无效的条件。解决这一问题的一种替代方法是使用求解$F(t)$的极小值的方法。由于已经有现成的求解$\vec{X}'(t)$的方法（例如，A.6 节中讨论的 Brent 的方法），求解极小值的方法可利用导数信息。或者仅仅利用$X(t)$（例如 Powell 的方向集方法，也在 A.6 节中讨论）。

另一种替代的方法是将曲线细分，用折线（参见 A.8 节）来近似表示，然后计算 Y 与这些折线的距离，以此作为距离的近似值或者局部定位对最接近的点的搜索。在后一种情形中，最小值的求解方法可以应用于与在所有线段中具有最小距离的线段对应的曲线参数区间。

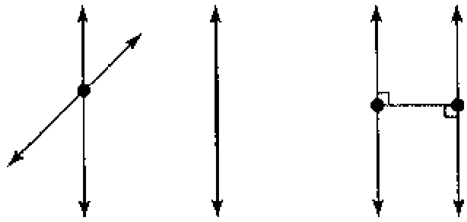
在细分之后，可通过计算 Y 与细分得到的折线之间的距离来计算距离平方。在细分中的最后一个子区间$[t_0, t_1]$内，方程（6.8）中的距离平方多项式$P(t)$的导数可对$[t_0, t_1]$上的根进行测试（参见 A.5 节中关于多项式的 Sturm 次序的子节）。如果没有根，那么$P(t)$在该区间上是单调的，并且距离的极大值和极小值出现在 t_0和t_1处。如果子区间是一个内部区间，那么距离的极小值不会出现在该子区间中。如果 t_0或t_1是原参数区间的端点，那么在这些点的距离的平方必须与计算所得的任何内部局部极值进行比较。如果$P'(t)$在子区间内有一个根，那么可用与对分法一样健壮的方法来定位根。如果$P'(t)$在子区间内有多个根，则进一步细分，直到子区间内最多只有一个根为止。

6.6　线形对象之间的距离

本节将介绍直线、射线和线段之间的两两组合（直线与直线、直线与射线、直线与线段、射线与射线，射线与线段和线段与线段）之间的距离。

6.6.1　直线到直线的距离

设直线表示为法线形式$\vec{n}_i \cdot X = c_i$，其中$i = 0, 1$。如果两条直线相交，则它们之间的距离为 0。否则，如果直线是平行的，而且它们不重合，那么它们之间的距离为正；如果它们是同一条直线，那么它们之间的距离为零。图 6.16 说明了这几种情形。



(a) (b)

图 6.16 几种直线—直线构形：（a）零距离；（b）正距离

对于直线平行的情形，它们之间的距离就是一条直线上的点P_0与另一条直线上的点$P_1 = P_0 + t\vec{n}_0$之间的距离。距离就是$\|t\vec{n}_0\|$。t值由$c_1 = \vec{n}_1 \cdot P_1 = \vec{n}_1 \cdot P_0 + t\vec{n}_1 \cdot \vec{n}_0$确定，其中$t = (c_1 - \vec{n}_1 \cdot P_0)/(\vec{n}_1 \cdot \vec{n}_0)$。在第一条线上的一个点为$P_0 = c_0\vec{n}_0/\|\vec{n}_0\|^2$。用它来替换方程中的$t$，将其代入$\|t\vec{n}_0\|$，并重新排列各项，可得如下的距离公式

$$\text{Distance}\left(\mathcal{L}_0, \mathcal{L}_1\right) = \begin{cases} 0, & \vec{n}_0 \cdot \vec{n}_1^{\perp} \neq 0 \\ \frac{|(\vec{n}_0 \cdot \vec{n}_0)c_1 - (\vec{n}_0 \cdot \vec{n}_1)c_0|}{\|\vec{n}_0\| |\vec{n}_0 \cdot \vec{n}_1|}, & \vec{n}_0 \cdot \vec{n}_1^{\perp} = 0 \end{cases} \tag{6.9}$$

如果$\|\vec{n}_0\| = \|\vec{n}_1\|$，则距离公式的第二部分可简化为$|c_1 - \sigma c_0|/\|\vec{n}_0\|$，其中$\sigma = \text{Sign}(\vec{n}_0 \cdot \vec{n}_1)$。如果有额外条件$\|\vec{n}_0\| = 1$，则可以避免除法运算。

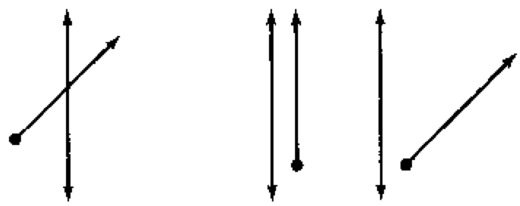
参数形式$P_i + t_i\vec{d}_i$（$i = 0, 1$）的等价距离公式为

$$\text{Distance}\left(\mathcal{L}_0, \mathcal{L}_1\right) = \begin{cases} 0, & \vec{d}_0 \cdot \vec{d}_1^{\perp} \neq 0 \\ \frac{|\vec{d}_0^{\perp} \cdot \vec{\Delta}|}{\|\vec{d}_0\|}, & \vec{d}_0 \cdot \vec{d}_1^{\perp} = 0 \end{cases} \tag{6.10}$$

其中$\vec{\Delta} = P_1 - P_0$。公式的第二部分为$\vec{\Delta}$在与给定的两条直线垂直的法线直线上投影的长度。

6.6.2 直线到射线的距离

这种距离的计算与直线之间距离的计算类似。惟一的区别是如果直线与射线是不平行的，则射线可能与直线不相交。图 6.17 显示了这种可能性。



(a) (b)

图 6.17 几种直线—射线构形：（a）零距离；（b）正距离

设直线$\mathcal{L}$的法线表示为$\vec{n}_0 \cdot X = c_0$。设射线$\mathcal{R}$的参数表示为$P_1 + t\vec{d}_1$（$t \geqslant 0$）。如果P_1在直线上$\vec{n}_0$所指向的一侧，那么，如果射线指向直线，直线与射线相交，即如果$\vec{n}_0 \cdot \vec{d}_1 < 0$，则直线与射线相交。否则，如果射线指向背离直线，它们之间的距离就是射线上的点P_1与其在直线上的投影（设为点P_0）之间的距离。类似地，如果P_1在直线上$\vec{n}_0$所指向的另一侧，那么，如果射线指向直线，则直线与射线相交，即如果$\vec{n}_0 \cdot \vec{d}_1 > 0$，则直线与射线相交。否

则，如果射线指向背离直线，则它们之间的距离就是射线上的点P_1与其在直线上的投影（设为点P_0）之间的距离。在不相交的情形中，如果$\vec{\Delta} = P_1 - P_0$，则直线与射线之间的距离为$|\vec{n}_0 \cdot \vec{\Delta}|/\|\vec{n}_0\| = |\vec{n}_0 \cdot P_1 - c_0|/\|\vec{n}_0\|$。因此在计算距离时，必须确定地计算出$P_0$。距离公式可概括为

$$\text{Distance}\,(\mathcal{L}, \mathcal{R}) = \begin{cases} 0, & (\vec{n}_0 \cdot \vec{d}_1)(\vec{n}_0 \cdot P_1 - c_0) < 0 \\ \frac{|\vec{n}_0 \cdot P_1 - c_0|}{\|\vec{n}_0\|}, & (\vec{n}_0 \cdot \vec{d}_1)(\vec{n}_0 \cdot P_1 - c_0) \geqslant 0 \end{cases} \quad (6.11)$$

对于直线的参数形式$P_0 + t_0\vec{d}_0$（$t_0 \in \mathbb{R}$）和射线的参数形式$P_1 + t_1\vec{d}_1$（$t_1 \geqslant 0$），等价的距离公式为

$$\text{Distance}\,(\mathcal{L}, \mathcal{R}) = \begin{cases} 0, & (\vec{d}_0^{\perp} \cdot \vec{d}_1)(\vec{d}_0^{\perp} \cdot \vec{\Delta}) < 0 \\ \frac{|\vec{d}_0^{\perp} \cdot \vec{\Delta}|}{\|\vec{d}_0\|}, & (\vec{d}_0^{\perp} \cdot \vec{d}_1)(\vec{d}_0^{\perp} \cdot \vec{\Delta}) \geqslant 0 \end{cases} \quad (6.12)$$

其中$\vec{\Delta} = P_1 - P_0$且$(x, y)^{\perp} = (y, -x)$。公式的第二部分为$\vec{\Delta}$在与给定的两条直线垂直的法线直线上的投影长度。

6.6.3 直线到线段的距离

设直线$\mathcal{L}$的法线表示为$\vec{n} \cdot X = c$，线段S的端点为Q_0和Q_1，它们或者相交，或者不相交。如果相交，则它们之间的距离为零。如果不相交，则它们之间的距离由离直线较近的一个端点决定。图 6.18 显示了这些情形。距离为

$$\text{Distance}\,(\mathcal{L}, S) = \begin{cases} 0, & (\vec{n} \cdot Q_0 - c)(\vec{n} \cdot Q_1 - c) < 0 \\ \min\left(\frac{|\vec{n} \cdot Q_0 - c|}{\|\vec{n}\|}, \frac{|\vec{n} \cdot Q_1 - c|}{\|\vec{n}\|}\right), & (\vec{n} \cdot Q_0 - c)(\vec{n} \cdot Q_1 - c) \geqslant 0 \end{cases} \quad (6.13)$$

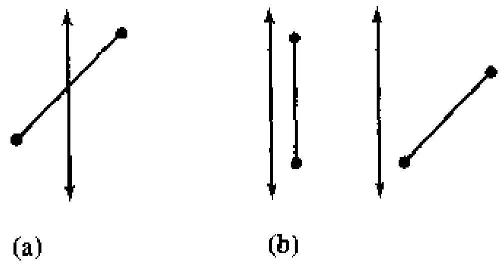


图 6.18　几种直线－线段构形：（a）零距离；（b）正距离

对于参数形式的直线$P_0 + t_0\vec{d}_0$（$t_0 \in \mathbb{R}$）和参数形式的射线$P_1 + t_1\vec{d}_1$（$t_1 \in [0, T_1]$），等价的距离公式为

$$\text{Distance}\,(\mathcal{L}, S) = \begin{cases} 0, & (\vec{d}_0^{\perp} \cdot \vec{\Delta})(\vec{d}_0^{\perp} \cdot (\vec{\Delta} + T_1\vec{d}_1)) < 0 \\ \min\left(\frac{|\vec{d}_0^{\perp} \cdot \vec{\Delta}|}{\|\vec{d}_0\|}, \frac{|\vec{d}_0^{\perp} \cdot (\vec{\Delta} + T_1\vec{d}_1)|}{\|\vec{d}_0\|}\right), & (\vec{d}_0^{\perp} \cdot \vec{\Delta})(\vec{d}_0^{\perp} \cdot (\vec{\Delta} + T_1\vec{d}_1)) \geqslant 0 \end{cases} \quad (6.14)$$

其中$\vec{\Delta} = P_1 - P_0$。

6.6.4 射线到射线的距离

设射线为$P_i + t_i\vec{d}_i$（$t_i \geqslant 0$，$i = 0, 1$）。如果射线相交，则它们之间的距离为零。如果

射线不相交，则最小距离可由如下方式确定：（1）一条射线的端点与另一条射线的内点；（2）两条射线的端点。我们首先考虑射线不平行的情形。图 6.19 说明了各种的可能性。图 6.19（a）显示了相交的射线，两条射线上的公共内点确定了它们之间的零距离。图 6.19（b）显示了一条射线的端点与另一条射线的内点确定的正距离。图 6.19（c）显示了两条射线的端点确定的正距离。

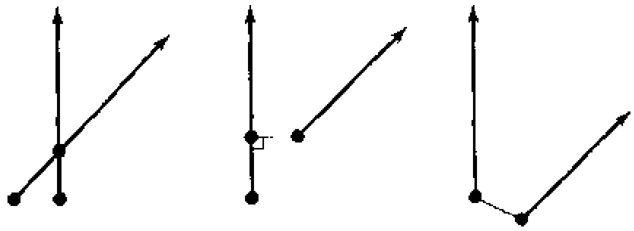


图 6.19　几种不平行的射线－射线构形：（a）零距离；（b）从端点到内部点的正距离；（c）从端点到端点的正距离

定义$\vec{\Delta} = P_0 - P_1$。任何两个点$P_0 + t_0\vec{d}_0$与$P_1 + t_1\vec{d}_1$之间距离的平方为

$$F(t_0, t_1) = \|t_0\vec{d}_0 - t_1\vec{d}_1 + \vec{\Delta}\|^2 = a_{00}t_0^2 - 2a_{01}t_0t_1 + a_{11}t_1^2 + 2b_0t_0 - 2b_1t_1 + c \qquad (6.15)$$

其中$a_{ij} = \vec{d}_i \cdot \vec{d}_j$, $b_i = \vec{d}_i \cdot \vec{\Delta}$且$c = \vec{\Delta} \cdot \vec{\Delta}$。$F$ 是一个非负的二次多项式。如果直线不平行，则它们必然相交于一个点，并且两条直线之间距离的平方为零（由于交点是两条线的公共点）。即存在参数$(\bar{t}_0，\bar{t}_1)$满足$F(\bar{t}_0，\bar{t}_1) = 0$。同时注意，零是 F 的全局极小值，因此在极小值处，梯度必然为零：

$$(0,0) = \vec{\nabla}F(\bar{t}_0, \bar{t}_1) = \left(2(\bar{t}_0\vec{d}_0 - \bar{t}_1\vec{d}_1 + \vec{\Delta}) \cdot \vec{d}_0, -2(\bar{t}_0\vec{d}_0 - \bar{t}_1\vec{d}_1 + \vec{\Delta}) \cdot \vec{d}_1\right) \qquad (6.16)$$

虽然这是一个可用标准方法求解的具有两个未知数的两个方程的线性系统，但基于如下的事实，可以采用一种更快速的求解方法。由于这两条直线不平行，因此向量$\vec{d}_0$和$\vec{d}_1$是线性无关的。方程（6.16）表明$\bar{t}_0\vec{d}_0 - \bar{t}_1\vec{d}_1 + \vec{\Delta}$是一个同时垂直于$\vec{d}_0$和$\vec{d}_1$的向量。在一个平面内，一个向量可同时垂直于两个线性无关的向量的惟一条件是该向量为零向量。因此，$\bar{t}_0\vec{d}_0 - \bar{t}_1\vec{d}_1 + \vec{\Delta} = \vec{0}$。用$\vec{d}_0^{\perp}$和$\vec{d}_1^{\perp}$点乘方程的两边，可得如下的解

$$(\bar{t}_0, \bar{t}_1) = \frac{(\vec{d}_1^{\perp} \cdot \vec{\Delta}, \vec{d}_0^{\perp} \cdot \vec{\Delta})}{\vec{d}_1^{\perp} \cdot \vec{d}_0} \qquad (6.17)$$

F 的阶层曲线为中心为$(\bar{t}_0, \bar{t}_1)$的椭圆。如果两线是平行的，那么对于任何t_0，F 都为常数，因此沿着整条直线，F 在$\partial F/\partial t_0 = 0$处取极小值

$$(\bar{t}_0, \bar{t}_1) = \left(\frac{a_{01}\bar{t}_1 - b_0}{a_{00}}, \bar{t}_1\right) \qquad (6.18)$$

F 的阶层曲线为平行于该直线的直线。F 在其定义域$[0,\infty)^2$的极小值情况取决于分析 F 的阶层曲线和其定义域之间的关系。

首先考虑不平行的射线。如果$\bar{t}_0 > 0$且$\bar{t}_1 > 0$，那么这两条射线相交于内点。如果$\bar{t}_0 > 0$且$\bar{t}_1 \leqslant 0$，那么 F 的极小值必然出现于$(\max\{\hat{t}_0, 0\}, 0)$，其中$\partial F(\hat{t}_0, 0)/\partial t_0 = 2(a_{00}\hat{t}_0 + b_0) = 0$。

考虑到 F 的阶层曲线仅仅接触 t_0 轴，这是非常清楚的。图 6.20 说明了这一点。注意，$\hat{t}_0=-b_0/a_{00}$ 且 $F(\hat{t}_0,0)=c-b_0^2/a_{00}=(\vec{d}_0^{\perp}\cdot\vec{\Delta})^2/\|\vec{d}_0\|^2$。类似地，如果 $\bar{t}_0\leqslant 0$ 且 $\bar{t}_1>0$，那么 F 的极小值必须出现于 $(0,\max\{\hat{t}_1,0\})$，其中 $\partial F(0,\hat{t}_1)/\partial t_1=2(a_{11}\hat{t}_1-b_1)=0$。注意，$\hat{t}_1=b_1/a_{11}$ 且 $F(0,\hat{t}_1)=c-b_1^2/a_{00}=(\vec{d}_1^{\perp}\cdot\vec{\Delta})/\|\vec{d}_1\|^2$。如果 $\bar{t}_0\leqslant 0$ 且 $\bar{t}_1\leqslant 0$，那么 F 的极小值可出现在其参数定义域的任一边界上，这取决于 F 的阶层曲线相对于边界的位置。然而，在这种情形中，不可能出现 $\partial F(\hat{t}_0,0)/\partial t_0=0$ 和 $\partial F(0,\hat{t}_1)/\partial t_1=0$，因此，分别检查每一个位置就足够了。下面给出了距离公式。假定只有当其他项的布尔表达式都检查过之后，最后一项才用于该距离公式。

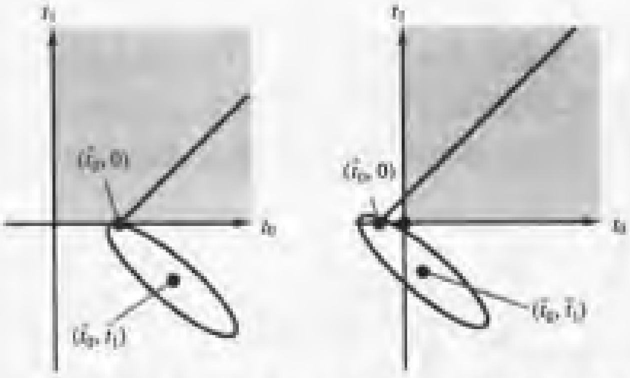


图 6.20　阶层曲线 F 与位于 $(\hat{t}_0,0)$ 或（0，0）的最小边界的关系

$$\text{Distance}\,(\mathcal{R}_0,\mathcal{R}_1)=\begin{cases}0, & \bar{t}_0>0 \text{ 且 } \bar{t}_1>0\\ |\vec{d}_0^{\perp}\cdot\vec{\Delta}|/\|\vec{d}_0\|, & \bar{t}_0>0 \text{ 且 } \bar{t}_1\leqslant 0\\ |\vec{d}_1^{\perp}\cdot\vec{\Delta}|/\|\vec{d}_1\|, & \bar{t}_1>0 \text{ 且 } \bar{t}_0\leqslant 0\\ \|\vec{\Delta}\|, & \text{其他情况}\end{cases}\qquad(6.19)$$

现在考虑射线平行的情形。图 6.21 显示了不同构形。图 6.21（a）显示了射线指向同一方向的情形。射线之间的距离由一条射线的端点和另一条射线的一个内点确定。图 6.21（b）显示了两条射线指向相反的方向，且两条射线与另一条射线重叠的情形（如果相互投影到对方）。射线之间的距离也由一条射线的端点和另一条射线的一个内点确定。图 6.21（c）显示了两条射线指向相反的方向，且两条射线没有投影重叠的情形。射线之间的距离由两条射线的两个端点确定。距离为

$$\text{Distance}\,(\mathcal{R}_0,\mathcal{R}_1)=\begin{cases}\|\vec{\Delta}\|, & \vec{d}_0\cdot\vec{d}_1<0 \text{ 且 } \vec{d}_0\cdot\vec{\Delta}\geqslant 0\\ |\vec{d}_0^{\perp}\cdot\vec{\Delta}|/\|\vec{d}_0\|, & \text{其他情况}\end{cases}\qquad(6.20)$$

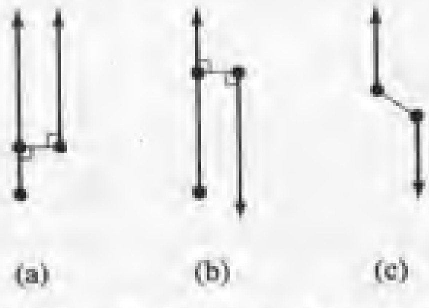


图 6.21　几种平行的射线—射线构形：（a）射线指向同一方向；（b）射线指向相反方向且并行重叠；（c）射线指向相反方向且不并行重叠

6.6.5 射线到线段的距离

设射线为 $P_0+t_0\vec{d}_0$（$t_0\geqslant 0$），并设线段为 $P_1+t_1\vec{d}_1$（$t_1\in[0,T_1]$）。这种结构类似于两条射线的情形，我们当时分析了对于指定的问题，F 的阶层曲线在整个 $\mathbb{R}^2$ 上如何与其定义域交互。相关的两条射线的边界点为 $(\hat{t}_0,0)$ 和 $(0,\hat{t}_1)$，F 在这些点的偏导数为零。对于射线与线段之间的距离问题，需要考虑的一个额外点是 $(\tilde{t}_0,T_1)$，该点满足 $\partial F(\tilde{t}_0,T_1)/\partial t_0=0$。其解为 $\tilde{t}_0=(a_{01}T_1-b_0)/a_{00}$。注意 $F(\tilde{t}_0,T_1)=a_{11}T_1^2-2b_1T_1+c-(a_{01}T_1-b_0)^2/a_{00}=(\vec{d}_0^{\perp}\cdot(\vec{\Delta}-T_1\vec{d}_1))^2/\|\vec{d}_0\|^2$。最后一个等式刚好说明我们计算的就是射线与端点为 $P_1+T_1\vec{d}_1$ 的线段之间距离的平方。

对于不平行的情形，如果 $(\bar{t}_0,\bar{t}_1)\in(0,\infty)\times(0,T_1)$，那么射线与线段相交于一个内点。否则，必须确定中心为 $(\bar{t}_0,\bar{t}_1)$ 的椭圆形阶层曲线首先在何处与定义域的边界相汇合。距离公式在下面给出。假定只有当其他项的布尔表达式都检查过之后，最后一项才用于该距离公式。

$$\text{Distance}(\mathcal{R},S)=\begin{cases}0, & \bar{t}_0>0\ \text{且}\ \bar{t}_1\in(0,T_1)\\ |\vec{d}_0^{\perp}\cdot\vec{\Delta}|/\|\vec{d}_0\|, & \hat{t}_0>0\ \text{且}\ \bar{t}_1\leqslant 0\\ |\vec{d}_0^{\perp}\cdot(\vec{\Delta}-T_1\vec{d}_1)|/\|\vec{d}_0\|, & \tilde{t}_0>0\ \text{且}\ \bar{t}_1\geqslant T_1\\ |\vec{d}_1^{\perp}\cdot\vec{\Delta}|/\|\vec{d}_1\|, & \hat{t}_1\in(0,T_1)\ \text{且}\ \bar{t}_0\leqslant 0\\ \|\vec{\Delta}\|, & \hat{t}_0\leqslant 0\ \text{且}\ \hat{t}_1\leqslant 0\\ \|\vec{\Delta}-T_1\vec{d}_1\|, & \tilde{t}_0\leqslant 0\ \text{且}\ \hat{t}_1\geqslant T_1\end{cases}\tag{6.21}$$

当射线与线段相交时，第一个方程成立，即距离为零；当线段的端点 P_1 与射线的一个内点最接近时，第二个方程成立；当线段的端点 $P_1+T_1\vec{d}_1$ 与射线的一个内点最接近时，第三个方程成立；当射线的端点 P_0 与线段的一个内点最接近时，第四个方程成立；当射线的端点 P_0 与线段的一个端点 P_1 最接近时，第五个方程成立；当射线的端点 P_0 与线段的一个端点 $P_1+T_1\vec{d}_1$ 最接近时，第六个方程成立。

对于平行的情形，距离为

$$\text{Distance}(\mathcal{R},S)=\begin{cases}\|\vec{\Delta}\|, & \vec{d}_0\cdot\vec{d}_1<0\ \text{且}\ \vec{d}_0\cdot\vec{\Delta}\geqslant 0\\ \|\vec{\Delta}-T_1\vec{d}_1\|, & \vec{d}_0\cdot\vec{d}_1>0\ \text{且}\ \vec{d}_0\cdot(\vec{\Delta}-T_1\vec{d}_1)\geqslant 0\\ |\vec{d}_0^{\perp}\cdot\vec{\Delta}|/\|\vec{d}_0\|, & \text{否则}\end{cases}\tag{6.22}$$

当射线与线段的方向相反，且线段在包含射线的直线上的投影与射线不相连时，第一个方程成立。当射线与线段的方向相同且线段在包含射线的直线上的投影与射线不相连时，第二个方程成立。当线段在包含射线的直线上的投影与射线相交时，第三个方程成立。

6.6.6 线段到线段的距离

设线段为 $P_i+t_i\vec{d}_i$ 其中 $t_i\in[0,T_i]$，其距离的计算方式和射线与线段之间距离的计算方式相似。但是，相关的另一个边界点是 $(T_0,\tilde{t}_1)$，其中 $\partial F/\partial t_1=0$。其解为 $\tilde{t}_1=(a_{01}T_0+b_1)/a_{11}$。注意，$F(T_0,\tilde{t}_1)=a_{00}T_0^2+2b_0T_0+c-(a_{01}T_0+b_1)^2/a_{11}=(\vec{d}_1^{\perp}\cdot(\vec{\Delta}+T_0\vec{d}_0))^2/\|\vec{d}_1\|^2$。

对于不平行的情形，如果 $(\bar{t}_0,\bar{t}_1)\in(0,T_0)\times(0,T_1)$，那么两条线段相交于一个内点。否则，必须确定中心为 $(\bar{t}_0,\bar{t}_1)$ 的椭圆形阶层曲线首先在何处与定义域的边界相汇合。距离公式在下面给出。假定只有当其他项的布尔表达式都检查过之后，最后四项才用于该距离公式。

$$
\text{Distance}\left(S_0, S_1\right)=\begin{cases}
0, & \bar{t}_0 \in (0, T_0) \text{ 且 } \bar{t}_1 \in (0, T_1) \\
|\vec{d}_0^{\perp} \cdot \vec{\Delta}|/\|\vec{d}_0\|, & \hat{t}_0 \in (0, T_0) \text{ 且 } \hat{t}_1 \leqslant 0 \\
|\vec{d}_0^{\perp} \cdot (\vec{\Delta} - T_1\vec{d}_1)|/\|\vec{d}_0\|, & \hat{t}_0 \in (0, T_0) \text{ 且 } \hat{t}_1 \geqslant T_1 \\
|\vec{d}_1^{\perp} \cdot \vec{\Delta}|/\|\vec{d}_1\|, & \hat{t}_1 \in (0, T_1) \text{ 且 } \hat{t}_0 \leqslant 0 \\
|\vec{d}_1^{\perp} \cdot (\vec{\Delta} + T_0\vec{d}_0)|/\|\vec{d}_1\|, & \hat{t}_1 \in (0, T_1) \text{ 且 } \hat{t}_0 \geqslant T_0 \\
\|\vec{\Delta}\|, & \hat{t}_0 \leqslant 0 \text{ 且 } \hat{t}_1 \leqslant 0 \\
\|\vec{\Delta} + T_0\vec{d}_0\|, & \hat{t}_0 \geqslant T_0 \text{ 且 } \hat{t}_1 \leqslant 0 \\
\|\vec{\Delta} - T_1\vec{d}_1\|, & \hat{t}_0 \leqslant 0 \text{ 且 } \hat{t}_1 \geqslant T_1 \\
\|\vec{\Delta} + T_0\vec{d}_0 - T_1\vec{d}_1\|, & \hat{t}_0 \geqslant T_0 \text{ 且 } \hat{t}_1 \geqslant T_1
\end{cases} \tag{6.23}
$$

当两条线段相交时，第一个方程成立，即距离为零；当第一条线段的一个内点与第二条线段的端点 P_1 最接近时，第二个方程成立；当第一条线段的一个内点与第二条线段的端点 $P_1 + T_1\vec{d}_1$ 最接近时，第三个方程成立；当第二条线段的一个内点与第一条线段的端点 P_0 最接近时，第四个方程成立；当第二条线段的一个内点与第一条线段的端点 $P_0 + T_0\vec{d}_0$ 最接近时，第五个方程成立；当第一条线段的的端点 P_0 与第二条线段的端点 P_1 最接近时，第六个方程成立；当第一条线段的的端点 $P_0 + T_0\vec{d}_0$ 与第二条线段的端点 P_1 最接近时，第七个方程成立；当第一条线段的的端点 P_0 与第二条线段的端点 $P_1 + T_1\vec{d}_1$ 最接近时，第八个方程成立；当第一条线段的的端点 $P_0 + T_0\vec{d}_0$ 与第二条线段的端点 $P_1 + T_1\vec{d}_1$ 最接近时，第九个方程成立。

对于平行的情形，距离为

$$
\text{Distance}\left(S_0, S_1\right)=\begin{cases}
\|\vec{\Delta}\|, & \vec{d}_0 \cdot \vec{d}_1 < 0 \text{ 且 } \vec{d}_0 \cdot \vec{\Delta} \geqslant 0 \\
\|\vec{\Delta} + T_0\vec{d}_0\|, & \vec{d}_0 \cdot \vec{d}_1 > 0 \text{ 且 } \vec{d}_0 \cdot (\vec{\Delta} + T_0\vec{d}_0) \geqslant 0 \\
\|\vec{\Delta} - T_1\vec{d}_1\|, & \vec{d}_0 \cdot \vec{d}_1 > 0 \text{ 且 } \vec{d}_0 \cdot (\vec{\Delta} - T_1\vec{d}_1) \geqslant 0 \\
\|\vec{\Delta} + T_0\vec{d}_0 - T_1\vec{d}_1\|, & \vec{d}_0 \cdot \vec{d}_1 < 0 \text{ 且 } \vec{d}_0 \cdot (\vec{\Delta} + T_0\vec{d}_0 - T_1\vec{d}_1) \geqslant 0 \\
|\vec{d}_0^{\perp} \cdot \vec{\Delta}|/\|\vec{d}_0\|, & \text{否则}
\end{cases} \tag{6.24}
$$

前面的四个方程与方程（6.23）的后面四个方程的表现形式相同。当线段在包含另一条线段的直线上的投影与该线段相交时，第五个方程成立。

6.7　线形对象到折线或多边形的距离

直线与多边形对象或折线之间的距离可用相同的算法来处理。如果直线与该对象不相交，则它们之间的距离为正，且必然由该对象的一个顶点与直线之间的距离来确定。分析该顶点与直线之间的距离就足够了。设顶点为 P_i，$0 \leqslant i < n$，直线表示为 $\hat{n} \cdot X = c$，其中 $\hat{n}$ 为单位长度向量。如果所有的 $\hat{n} \cdot P_i - c > 0$ 或所有的 $\hat{n} \cdot P_i - c < 0$，那么对象完全位于直线的同一侧，此时距离为 $\min_i |\hat{n} \cdot P_i - c|$。否则，必然存在两个连续的点，$P_i$ 和 P_{i+1}，使得 $(\hat{n} \cdot P_i - c)(\hat{n} \cdot P_{i+1} - c) \leqslant 0$，且对象与直线相交。在这种情形下，直线与对象之间的距离为零。

对于设定不是一个区域的边界的一条开放的折线或闭合的折线，一条射线或线段与折线之间的距离可通过标准的方式来求得，即逐一计算射线或线段与折线的每一条线段之间的距离，然后取其中的最小值。

射线与一个实心多边形之间的距离，同样可以通过逐一计算射线与多边形的每一条边之间的距离，并选取最小距离来计算。一点细小的修改可能允许早一点退出算法。点在多边形内的测试（参看 13.3 节）可用于射线的原点。如果该点在多边形内，那么射线与多边形之间的距离为零。如果该点在多边形之外，则重新进行逐一比较。

对于计算线段与多边形之间的距离来说，逐一计算比较是不够的。当线段完全在多边形内部时，就会出现问题。这时线段与多边形的任何边的距离都为正，但是线段与多边形之间的距离为零，因为线段包含在多边形之内。然而，我们可以对线段的端点进行点是否在多边形内的测试。如果两个端点都在多边形内，则距离为零。如果两个端点都在多边形之外，那么可以应用逐一计算比较的方法。

在计算线段与折线之间的距离时，与用于点与多边形之间距离的排除测试类似的并不费时的排除测试可能用于多边形边界的排除，但相对复杂一些。点与多边形的排除基于对在包含圆心位于测试点的圆的轴对齐无限带之外的选择，或者基于对在包含该圆的轴对齐矩形之外的线段的选择。在当前的讨论中，测试对象是线段S，而不是一个点。如果从S到已处理过的折线的各线段的距离的当前最小值为μ，那么，如果另一条折线线段在S产生的半径为μ的“舱”之外，则它不能使μ更新。正如圆是与测试点 Y 的距离为μ的点集一样，“舱”是与测试线段S的距离为μ的点集。该对象是一个具有半球形帽子的矩形。图 6.22 显示了作为图 6.4 中 Y 的模拟的线段S的构形。轴对齐无限带或轴对齐边界矩形可被构造并且可以用来进行选择，正如在计算点与折线之间的距离时所进行的一样。

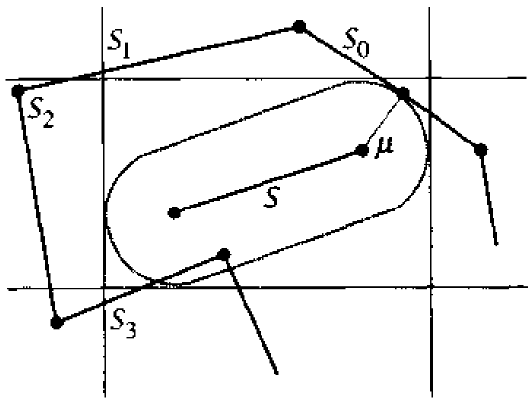


图 6.22　线段S获得当前最小距离μ的构形，μ类似于图 6.4 中点 Y 获得的当前最小距离

6.8　线形对象到二次曲线的距离

首先考虑计算直线与二次曲线之间距离的情形。如果直线与二次曲线相交，那么它们之间的距离为零。可使用直线的参数形式，$X(t)=P+t\vec{d}$，来测试直线与二次曲线之间的相交。二次曲线用$Q(X)=X^{\mathrm{T}}\mathbf{A}X+B^{\mathrm{T}}X+c=0$来隐含定义。将直线方程代入二次曲线方程得到如下的多项式方程

$$(\vec{d}^{\mathrm{T}}\mathbf{A}\vec{d})t^2+\vec{d}^{\mathrm{T}}(2\mathbf{A}P+B)t+(P^{\mathrm{T}}\mathbf{A}P+B^{\mathrm{T}}P+C)=e_2t^2+e_1t+e_0=0$$

当 $e_1^2-4e_0e_2\geqslant 0$ 时，上述方程具有实数解，此时直线与曲线之间的距离为零。

如果上述方程只有复数解，那么直线与曲线不相交，它们之间的距离为正数。在这种情形中，我们使用直线方程 $\hat{n}\cdot X=c, \|\hat{n}\|=1$ 来进行分析。要解决的问题是在二次曲线中找到一个点 X 使 $F(X)$ 取得极小值。这是一个具有限制条件的极小值问题，可使用拉格朗日乘子（参见 A.9.3 节）来求解。定义

$$G(X,s)=(\hat{n}\cdot X-c)^2+sQ(X)$$

当 $\vec{\nabla}G=\vec{0}$ 且 $\partial G/\partial s=0$ 时，G 取极小值。第一个方程为 $2(\hat{n}\cdot X-c)\hat{n}+s\vec{\nabla}Q=\vec{0}$，并且第二个方程仅仅重复了限制条件 $Q=0$。用 $\vec{d}=\hat{n}^{\perp}$ 与方程的两边点乘，得到如下条件

$$L(X):=\vec{d}\cdot\vec{\nabla}Q(X)=\vec{d}\cdot(2\mathrm{A}\vec{X}+B)=0$$

这是一个关于 X 的线性方程。在几何意义上，条件 $\vec{d}\cdot\vec{\nabla}Q=0$ 意味着，当距离最小值为正时，连接最接近的点的线段必须同时垂直于直线和二次曲线。图 6.23 说明了这点。

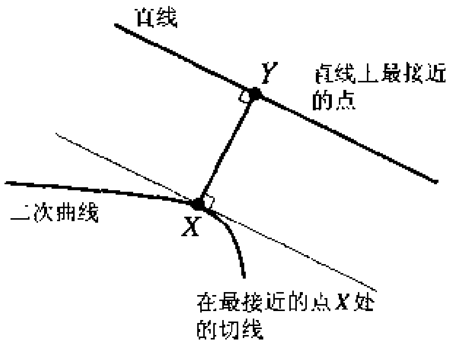


图 6.23 与最接近点相连的线段同时垂直于两个对象

现在剩下的问题是求解两个关于 X 的多项式方程 $L(X)=0$ 和 $Q(X)=0$。当 $\mathrm{A}\vec{d}=\vec{0}$ 时，线性方程退化。此时最可能出现的情况是，二次曲线将仅仅表示一条直线或一个点，虽然二次曲线也可能是一条抛物线或二次曲线。例如，由 $y=x^2$ 定义的抛物线与 $x=0$ 定义的直线将引起这种问题，但是直线与二次曲线的相交测试应该已经排除了这种可能性。退化的二次曲线方程所定义的直线与测试直线可能是不相交和平行的。在这种情况下，$\vec{d}\cdot B=0$ 且 $\mathrm{A}\vec{d}=\vec{0}$ 和 $L(X)=0$ 是同一方程，因此，应该使用求解两条直线之间的距离的算法来求它们之间的距离。

当 $\mathrm{A}\vec{d}\neq\vec{0}$ 时，可以求得该线性方程的一个变量，将这个变量代入二次曲线方程可以得到一个具有一个变量的二次多项式。这一方程很容易求解，参见 A.2 节。它的解 X 用于计算距离 $|\hat{n}\cdot X-c|$。

计算直线与二次曲线之间的距离的另一种方法，是使用一种数值极小值算法。如果直线为 $X(t)=P+t\vec{d}\,(t\in\mathbb{R})$，点 X 与二次曲线之间的距离为 $F(X)$，那么直线上的一个点 $X(t)$ 与二次曲线之间的距离为 $G(t)=F(P+td)$。极小值算法可通过如下方法来实现：搜索 t 的定义域 $\mathbb{R}$，以得到产生 $G(t)$ 的极小值的 t。需要考虑的折中是双重的。如果排除变量以得到高次数的单变量多项式方程，那么建立多项式方程系统的方法具有潜在的数值问题。排除过程和根的寻找都可能由于接近于零的参数而存在数值错误。建立求解方程极小值的函数可能在数值上更加稳定，但是，如果最初的猜测不接近于极小值点，那么集中地求解极

小值将产生速度慢的问题，或者产生迭代局限于局部极小值而得不到全局极小值的问题。

上面的讨论涉及一条直线和一条曲线。如果线形对象是一条射线，就需要对算法做一点修改。首先，计算包含射线的直线与曲线之间的距离。假定 Y 是直线上最接近于曲线的点，那么对某些 t 有 $Y = P + t\vec{d}$。如果 $t \geqslant 0$，那么 Y 在射线上，且射线与曲线之间的距离和直线与曲线之间的距离相同。但是，如果 $t < 0$，那么直线上最接近于曲线的点不在射线上。在这种情形中，应该利用 6.4 节中介绍的方法来计算射线的原点 P 与曲线之间的距离，即Distance(P, C)，其中 C 表示曲线。射线与曲线之间的距离就是 Distance(P, C)。

如果线形对象是一条线段，那么首先计算包含线段的直线与曲线之间的距离。假定 Y 是直线上最接近于曲线的点，那么对某些 t 有 $Y = P + t\vec{d}$。如果 $t \in [0, 1]$，则 Y 已经在线段上，且线段与曲线之间的距离是 Distance(Y, C)。但是，如果 $t < 0$，则线段与曲线之间的距离为 Distance(P, C)。如果 $t > 1$，则线段与曲线之间的距离为 Distance($P + \vec{d}, C$)。

6.9　线形对象到多项式曲线的距离

首先考虑计算一条直线与一条多项式曲线之间的距离的情形。设直线用 $\hat{n} \cdot X = c$ 来表示，其中 $\hat{n}$ 为单位向量。该直线与多项式曲线 $X(t)$（$t \in [t_0, t_1]$）之间的距离出现在使 $F(t) = (\hat{n} \cdot X(t) - c)^2$ 取极小值的 t 处。可直接对 $F(t)$ 应用数值极小值算法，或者，也可以利用微积分方法来求 $F'(t) = 0$ 的解，以此作为出现极小值的潜在位置。在后一种情形中，$F'(t) = 2(\hat{n} \cdot X(t) - c)(\hat{n} \cdot \vec{X}'(t))$，这是一个 $2n - 1$ 次多项式，其中 $X(t)$ 的次数为 n。可应用一种多项式的求根算法来求解该方程。利用变化的细分法可实现根的定位，正如在计算点与多项式曲线之间的距离时所做的一样。

如果线形对象是一条射线，那么需要对算法做一点修改。首先，计算包含射线的直线与曲线之间的距离。假定 Y 是直线上最接近于曲线的点，那么对某些 t 有 $Y = P + t\vec{d}$。如果 $t \geqslant 0$，那么 Y 在射线上，且射线与曲线之间的距离和直线与曲线之间的距离相同。但是，如果 $t < 0$，那么直线上最接近于曲线的点不在射线上。在这种情形中，应该利用 6.4 节中介绍的方法来计算射线的原点 P 与曲线之间的距离，即 Distance(P, C)，其中 C 表示曲线。射线与曲线之间的距离就是Distance(P, C)。

如果线形对象是一条线段，那么首先计算包含线段的直线与曲线之间的距离。假定 Y 是直线上最接近于曲线的点，那么对某些 t 有 $Y = P + t\vec{d}$。如果 $t \in [0, 1]$，那么 Y 已经在线段上，且线段与曲线之间的距离是Distance(Y, C)。但是，如果 $t < 0$，则线段与曲线之间的距离为Distance(P, C)。如果 $t > 1$，则线段与曲线之间的距离为 Distance($P + \vec{d}, C$)。

6.10　GJK 算法

我们现在来讨论一种计算二维空间中凸多边形之间的距离的高效算法。其最初的思想由 E.G.Gilbert，D.W.Johnson 和 S.S.Keerthi（1988）提出，用于处理三维空间中的凸多面体，但是该思想适用于任意维数空间中的一般凸多面体。该算法被演化成称为 GJK 算法的通用算法，其名称就是上述论文的三个作者的名字的首字母所构成的缩写。该算法后来被扩展，用于处理一般的凸形（Gilbert 和 Foo，1990）。该算法的一种改进也被提出，用于计算当多面体相交时多面体之间的渗透距离（Cameron，1997）。

6.10.1 集合运算

两个集合 A 和 B 的闵可夫斯基和（Minkowski sum）定义为由每一个集合中的一个向量组成的向量对的所有和的集合。用公式来表示就是，$A+B=\{X+Y:X\in A,\ Y\in B\}$。集合 B 的负集为 $-B=\{-X:X\in B\}$。集合的闵可夫斯基差（Minkowski difference）为 $A-B=\{X-Y:X\in A,\ Y\in B\}$。注意 $A-B=A+(-B)$。如果集合 A 和 B 是凸集，那么 $A+B$，$-B$ 和 $A-B$ 都是凸集。如果集合 A 是具有 n 个顶点的凸多边形，B 是具有 m 个顶点的凸多边形，那么在最糟糕的情形下，和 $A+B$ 具有 $n+m$ 个顶点。图 6.24 显示了 A 是一个三角形 $\langle U_0,U_1,U_2\rangle=\langle(0,0),(2,0),(0,2)\rangle$，$B$ 是一个三角形 $\langle V_0,V_1,V_2\rangle=\langle(2,2),(4,1),(3,4)\rangle$ 的情形。原点用黑点标识。

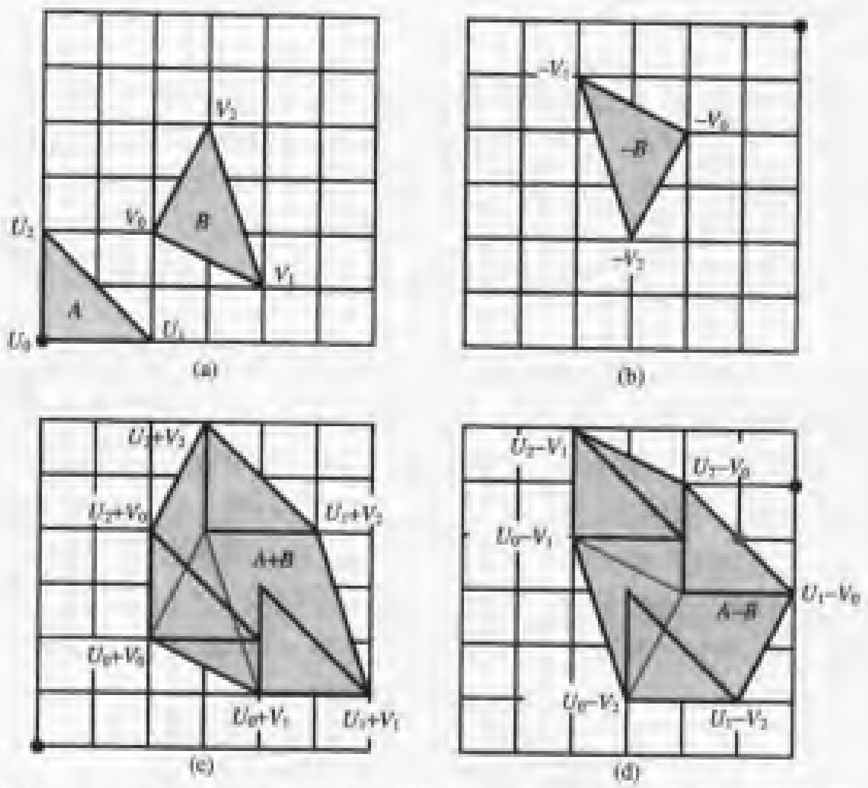


图 6.24 （a）三角形 A 和 B；（b）集合 $-B$；（c）集合 $A+B$；（d）集合 $A-B$。其中灰色点是集合 $A-B$ 中距原点最近的点，黑色点是原点（0，0）

图 6.24（a）显示了原三角形。图 6.24（b）显示了 $-B$。图 6.24（c）显示了 $A+B$。为了提供关于这种和的几何直观，该图显示了三个三角形，黑色的边对应于三角形 A，它由三角形 B 的三个顶点平移而形成。三角形 B 的边表示为灰色。想像一下，用平移的三角形 A 来画六边形的内部，在三角形 B 内移动 U_0+V_0 得到 A。相似的几何直观也表示在 $A-B$ 的画法中（如图 6.24（d）所示）。

任意两个集合 A 和 B 之间的距离可表示为

$$\mathrm{Distance}(A,B)=\min\{\|X-Y\|:X\in A,\ Y\in B\}=\min\{\|Z\|:Z\in A-B\}$$

后面的方程说明闵可夫斯基差在距离计算中起着非常重要的作用。最小距离由 $A-B$

内的一个点与原点之间的距离确定。图 6.24（d）将这个点显示为两个三角形。最接近于原点的点是位于$(-1,-1)\in A-B$的深灰色点。该点由$(1,1)\in A$和$(2,2)\in B$产生，因此 A 与 B 之间的距离为$\sqrt{2}$，并由上述的点确定。

距离计算的核心是如何有效地搜索 $A-B$ 以找到与原点最接近的点。一种直观的算法是直接计算 $A-B$，然后迭代计算各边并计算各边与原点之间的距离。这些距离的最小值就是 A 与 B 之间的距离。这种方法效率不高，因为计算作为点集 $U-V$ 的凸包的 $A-B$ 需要花费大量的时间，其中 U 是 A 的一个顶点，V 是 B 的一个顶点。而且，逐一搜索各边将处理一些从原点看不见的边。由于 A 有 n 个顶点，B 有 m 个顶点，因此该凸包可能有 nm 个顶点，所以这种方法是 O（nm）。GJK 是一种避免直接的凸包计算并将搜索定位于与原点接近的边的迭代方法。

6.10.2 算法概述

这里讨论的是一般的 n 维图形 A 和 B 的问题。设 $C=A-B$，其中 A 和 B 是凸集，C 自身也是一个凸集。如果$0\in C$，那么原来的两个集合相交并且它们之间的距离为零。否则，设$Z\in C$ 是最接近于原点的点。在几何上很清楚，只有一个这样的点存在，并且它必定位于 C 的边界上。然而，有许多的$X\in A$ 和$Y\in B$ 使得$X-Y=Z$。例如，二维空间中两个不相交的多边形，它们之间最接近的方式是各属于不同多边形的两条平行的边，这时就出现了这种情形。

GJK 算法是一种建立 C 的边界点次序的有效的降序方法，在这种次序中，每一个点与原点的距离都小于它前面的点。事实上，这种算法产生了一组排序的顶点在 C 上的单形（在二维空间中是三角形，在三维空间中是四面体），每一个单形与原点的距离都比其前一个单形小。设S_k 表示第 k 位的单形的顶点，$\bar{S}_k$ 表示单形自身，点$V_k\in\bar{S}_k$ 是从$\bar{S}_k$中选取的与原点最接近的点。初始时，$S_0=\emptyset$（空集），且V_0是 C 上的任意点。集合 C 通过 0 和方向V_0投影到直线上，得到的投影是闭合的，且在直线上包围一个区间。在投影线上最左边的区间端点由点$W_0\in C$ 产生。下一个单形顶点集合是$S_1=\{W_0\}$。图 6.25 说明了这一点。

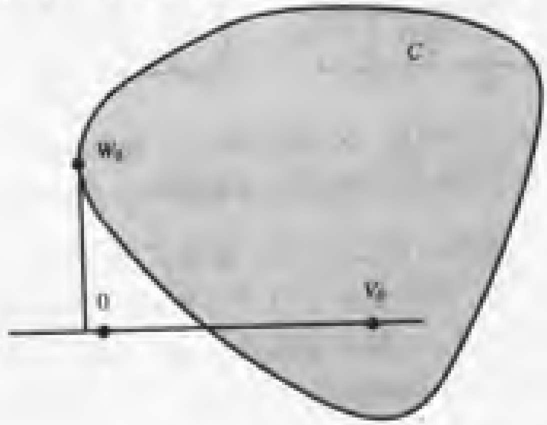


图 6.25　GJK 算法中的第一次迭代

由于S_1 是一个单独的点集，因此$\bar{S}_1=S_1$且$V_1=W_0$ 是$\bar{S}_1$上最接近于原点的点。集合 C 现在被投影到包含 0 且方向为V_1的直线上。在投影线上最左边的区间端点由点$W_1\in C$产生。下一个单形顶点集合是$S_2=\{W_0,W_1\}$。图 6.26 说明了这一点。

集合 $\bar{S}_2$ 是线段 (W_0, W_1)。$\bar{S}_2$ 上最接近于原点的点是边的内点V_2。集合 C 现在被投影到包含 0 且方向为 V_2 的直线上。在投影线上最左边的区间端点由点 $W_2 \in C$ 产生。下一个单形顶点集合是 $S_3 = \{W_0, W_1, W_2\}$。图 6.27 说明了这一点。

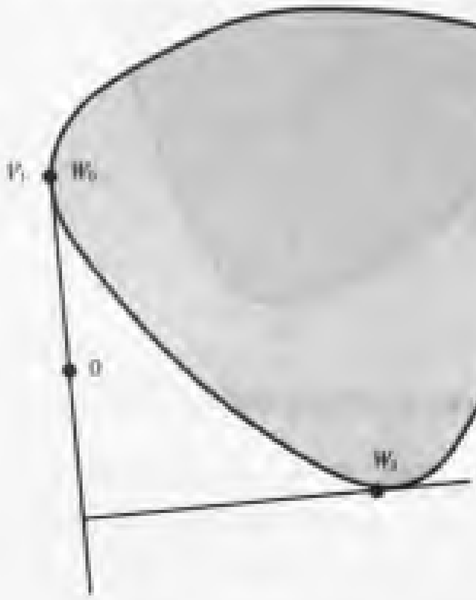


图 6.26　GJK 算法中的第二次迭代

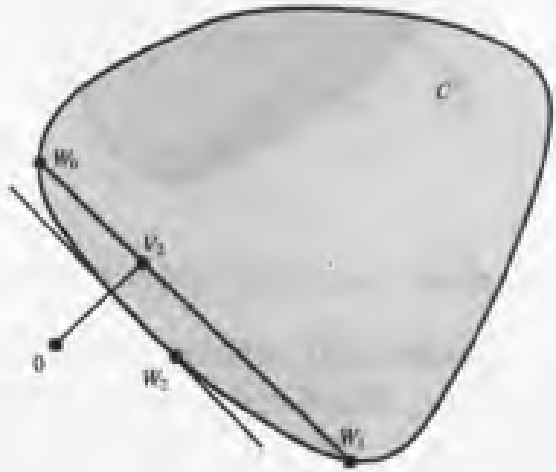


图 6.27　GJK 算法中的第三次迭代

集合$\bar{S}_3$是三角形(W_0, W_1, W_2)。$\bar{S}_3$上最接近于原点的点是边(W_0, W_2)上的点V_3。产生的下一个单形顶点是W_3。下一个单形顶点集合是$S_4 = \{W_0, W_2, W_3\}$。老的单形顶点W_1被抛弃。图 6.28 说明了这一点。单形$\bar{S}_3$用深灰色表示。

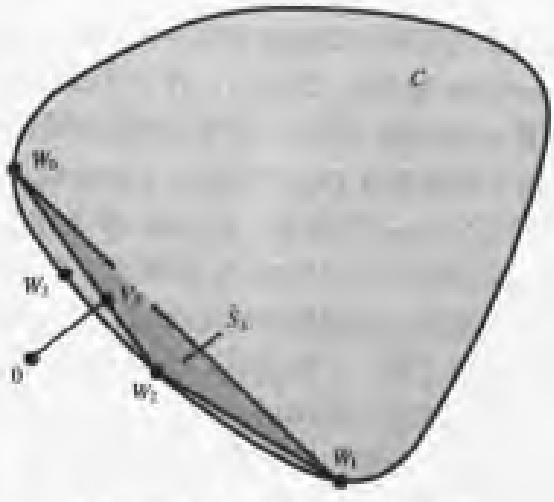


图 6.28　GJK 算法中的第四次迭代

一般地，V_{k+1}被选择作为凸包$S_k \cup \{W_k\}$中最接近原点的点。选择下一个单形顶点集合S_{k+1}，使$M \subseteq S_k \cup \{W_k\}$具有最少数量的元素，并使$V_{k+1}$在凸包 M 上。这样的 M 必须存在且惟一。图 6.29（a）显示了凸包 $S_3 \cup \{W_3\}$，这是一个四边形。下一次迭代V_4显示在该凸包上。图 6.29（b）显示了由$M = \{W_0, W_2, W_3\}$产生的单形$\bar{S}_4$。

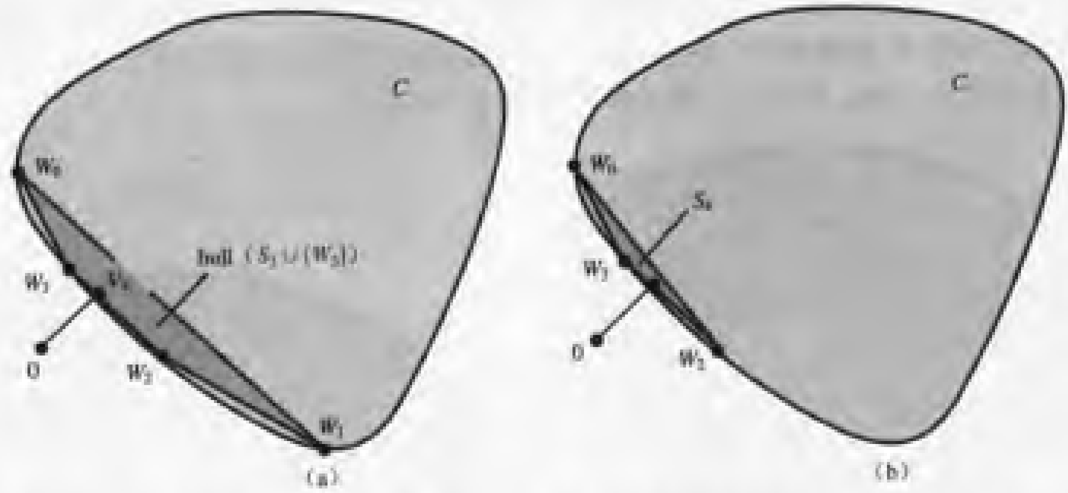


图 6.29　(a) 凸包 $S_k \cup \{W_k\}$ 中 V_{k+1} 的图解；(b) 从 $M = \{W_0, W_2, W_3\}$ 中生成的新单形 S_{k+1}

我们没有证明性地指出过迭代的次序是按长度单调递减的，即 $\|V_{k+1}\| \le \|V_k\|$。实际上，如果 $V_k = Z$，即在最近点时，等式才成立。对于凸面体对象，可以通过有限步的运算得到最近点。对于一般的凸形，次序可以是无限的，但必须收敛于 Z。如果 GJK 算法应用于这样的对象，必须应用某些种类的终止条件。如果算法在浮点数系统中实现，还可能产生数值问题。van den Bergen（1997，1999，2001a）给出了关于其中的缺陷的讨论，并且其中的思想已在一个名为 SOLID 的三维空间碰撞测试系统中实现（冲突检测软件库）（van den Bergen，2001b）。主要的问题是单形最终在一维或多维空间变成平的。

6.10.3 其他算法

GJK 算法并非是计算凸多边形或凸多面体之间距离的惟一算法，但是它具有公开开放的优秀而且健壮的实现（van den Bergen，2001b）。两个不相交的凸多边形之间的距离可使用旋转测径器的方法来计算（Pirzadeh 1999）。这种功能强大的方法对解决其他类型的计算几何问题也很有用。假定两个多面体有 $O(n)$ 个顶点，Cameron 和 Culley（1986）给出了一种计算距离的在时间和空间上为 $O(n^2)$ 的算法。Dobkin 和 Kirkpatrick（1990）给出了更好的算法，即在空间上为 $O(n)$，时间上为 $O(\log^2 n)$ 的算法。可是在共用域上没有出现这种算法的实现。这种方法的基础是建立多面体的层次表示，这对于解决其他问题（例如，在快速确定多面体在一个给定方向上的一个极端点）也很有用。最近，Lin-Canny 算法（Lin 和 Canny，1991）给出了一种在空间上为 $O(n)$，在时间上（根据经验）也是 $O(n)$ 的算法。该算法维护最接近的（多面体的）面对，以利用相关的坐标系。在计算一个坐标系中的距离之后，多面体移动少许距离，距离必须在下一个坐标系中重新计算。增加的更新在时间上为 $O(1)$。基于该方法的实现有 I-Collide（Cohen 等 1995）和 V-Clip（Mirtich 1997）。

第 7 章　二维相交

本章包含了计算二维几何图元相交的信息。最简单的对象组合是其中之一为线形对象（直线、射线、线段）的情形。本章前面四节介绍这类组合；7.5 节介绍两条二次曲线的相交；7.6 节介绍两条多次曲线的相交；最后一节介绍轴分离的方法，这是一种非常有效的处理凸形对象相交的技术。

7.1　线形对象之间的相交

先回忆一下第 5 章中对直线、射线和线段的定义。二维空间中直线的参数形式为 $P + t\vec{d}$，其中 $\vec{d}$ 为非零向量，$t \in \mathbb{R}$。射线的参数形式与此相同，不同的是 $t \in [0, \infty)$。点 P 为射线的原点。线段的参数形式也与此相同，不同的是 $t \in [0, 1]$。点 P 和 $P + \vec{d}$ 是线段的端点。线形对象是对直线、射线和线段的通称术语。

给定两条直线 $P_0 + s\vec{d}_0$ 和 $P_1 + t\vec{d}_1$，$s, t \in \mathbb{R}$，它们之间的关系只能是如下之一：相交、平行，以及为同一直线。为了便于确定其中的一种关系，我们定义二维空间中两个向量的数量值运算 $\text{Kross}((x_0, y_0), (x_1, y_1)) = x_0y_1 - x_1y_0$。这种运算与三维空间中的叉积相关，即 $(x_0, y_0, 0) \times (x_1, y_1, 0) = (0, 0, \text{Kross}((x_0, y_0), (x_1, y_1)))$。这种运算具有如下的性质：$\text{Kross}(\vec{u}, \vec{v}) = -\text{Kross}(\vec{v}, \vec{u})$。

如果交点存在，则可通过求解具有两个未知数的两个方程来得到交点，方程可通过设置 $P_0 + s\vec{d}_0 = P_1 + t\vec{d}_1$ 而建立。移项可得，$s\vec{d}_0 - t\vec{d}_1 = P_1 - P_0$。设 $\vec{\Delta} = P_1 - P_0$ 并应用 Kross 运算可得，$\text{Kross}(\vec{d}_0, \vec{d}_1)\, s = \text{Kross}(\vec{\Delta}, \vec{d}_1)$ 和 $\text{Kross}(\vec{d}_0, \vec{d}_1)\, t = \text{Kross}(\vec{\Delta}, \vec{d}_0)$。如果 $\text{Kross}(\vec{d}_0, \vec{d}_1) \neq 0$，则直线相交于一点，由 $s = \text{Kross}(\vec{\Delta}, \vec{d}_1)/\text{Kross}(\vec{d}_0, \vec{d}_1)$ 或 $t = \text{Kross}(\vec{\Delta}, \vec{d}_0)/\text{Kross}(\vec{d}_0, \vec{d}_1)$ 确定。如果 $\text{Kross}(\vec{d}_0, \vec{d}_1) = 0$，则两直线或者平行，或者是同一直线。如果有向向量的 Kross 运算为 0，那么上述的带 s 和 t 的方程退化为方程 $\text{Kross}(\vec{\Delta}, \vec{d}_0) = 0$，因为 $\vec{d}_1$ 是 $\vec{d}_0$ 的数量倍数。如果该方程成立，则两直线是同一直线；否则，两直线平行。

如果运用浮点算术运算来区分平行与不平行，则当 $\text{Kross}(\vec{d}_0, \vec{d}_1)$ 接近于 0 时，可能得到奇怪的结果。基于 Kross 与三维叉积的关系，用 Kross 来表示叉积的一个标准的恒等式为 $\|\text{Kross}(\vec{d}_0, \vec{d}_1)\| = \|\vec{d}_0\|\|\vec{d}_1\| |\sin\theta|$，其中 θ 是 $\vec{d}_0$ 与 $\vec{d}_1$ 之间的夹角。如果最后一个方程的右式接近于 0，则其三项中至少要有一项接近于 0。对于一些小的公差 $\varepsilon > 0$，使用绝对误差来进行比较的平行检测 $\|\text{Kross}(\vec{d}_0, \vec{d}_1)\| \leqslant \varepsilon$ 可能并不适合于一些应用。比如，用这种检测来判断长度很小的两个互相垂直的向量，就可能会得到平行的结果，但实际上它们却是垂直的。如果可能的话，应用程序应该要求用单位向量来表示直线的方向。这样，绝对误差检测就转化为两个方向之间夹角的正弦值检测：$\|\text{Kross}(\vec{d}_0, \vec{d}_1)\| = |\sin\theta| \leqslant \varepsilon$。对于足够小的角度，该检测是对角度的一个有效的阈值，因为对足够小的角度来说 $\sin\theta \doteq \theta$。如果应用程序不能满足用单位向量来表示直线方向，那么，应该基于相对误差来进行平行检测：

$$\frac{\|\text{Kross}(\vec{d}_0, \vec{d}_1)\|}{\|\vec{d}_0\|\|\vec{d}_1\|} = |\sin\theta| \leqslant \varepsilon$$

通过运用如下的等价不等式，可以避免两个长度的平方根和除法计算

$$\|\text{Kross}(\vec{d}_0, \vec{d}_1)\|^2 \leqslant \varepsilon^2\|\vec{d}_0\|^2\|\vec{d}_1\|^2$$

如果两个线形对象，一条直线($s \in \mathbb{R}$)和一条射线($t \geqslant 0$)，存在交点，则交点可通过上述求解 s 和 t 的方法来确定。然而，必须验证 $t \geqslant 0$。如果$t < 0$，则直线与包含射线的直线相交，但交点不在射线上。上述求解 t 的方法需进行除法运算。注意到$t = \text{Kross}(\vec{\Delta}, \vec{d}_0)\text{Kross}(\vec{d}_0, \vec{d}_1)/(\text{Kross}(\vec{d}_0, \vec{d}_1))^2$，并使用等价检测$\text{Kross}(\vec{\Delta}, \vec{d}_0)\text{Kross}(\vec{d}_0, \vec{d}_1) \geqslant 0$，可以得到一种能够避免这种代价的实现。如果实际的等价检测显示 $t \geqslant 0$，并且应用程序需要知道对应的交点，那么，仅需要直接计算 t，就可推迟计算除法，直到需要计算时才计算。当线形对象之一是射线或线段时，可用相同的方法来检测 s 和 t。

最后，当两个线形对象在同一条直线上时，线形对象将相交于一个 t 区间，区间可能为空、半无限或无限。计算相交的区间显得很繁琐，但并不复杂。作为一个例子，我们来考察如下的情形，两个线形对象都是线段，$s \in [0,1]$ 且$t \in [0,1]$。我们需要计算与 t 区间[0, 1]对应的第二条线段的 s 区间。第一个端点可表示为$P_1 = P_0 + s_0\vec{d}_0$，第二个端点可表示为$P_1 + \vec{d}_1 = P_0 + s_1\vec{d}_0$。如果$\vec{\Delta} = P_1 - P_0$，则$s_0 = \vec{d}_0 \cdot \vec{\Delta}/\|\vec{d}_0\|^2$且 $s_1 = s_0 + \vec{d}_0 \cdot \vec{d}_1/\|\vec{d}_0\|^2$。$s$ 区间为$[s_{min}, s_{max}] = [\min(s_0, s_1), \max(s_0, s_1)]$。相交的参数区间$[0,1] \cap [s_{min}, s_{max}]$可能为空集。线段相交的二维空间交点可通过将区间端点代入表达式$P_0 + s\vec{d}_0$而从相交区间中求得。求解两条直线相交的伪码列举如下。如果不相交，则函数的返回值为 0；如果只有一个交点，则返回值为 1；如果在同一直线上，则返回值为 2。只有当函数返回 1 时，返回的点 I 才有意义。

```
int FindIntersection(Point P0, Point D0, Point P1, Point D1, Point& I)
{
    // Use a relative error test to test for parallelism.  This effectively
    // is a threshold on the angle between D0 and D1.  The threshold
    // parameter 'sqrEpsilon' can be defined in this function or be
    // available globally.

    Point E = P1 - P0;
    float kross = D0.x * D1.y - D0.y * D1.x;
    float sqrKross = kross * kross;
    float sqrLen0 = D0.x * D0.x + D0.y * D0.y;
    float sqrLen1 = D1.x * D1.x + D1.y * D1.y;
    if (sqrKross > sqrEpsilon * sqrLen0 * sqrLen1) {
        // lines are not parallel
        float s = (E.x * D1.y - E.y *D1.x) / kross;
        I = P0 + s * D0;
        return 1;
    }

    // lines are parallel
    float sqrLenE = E.x * E.x + E.y * E.y;
    kross = E.x * D0.y - E.y * D0.x;
```

```
        sqrKross = kross * kross;
        if (sqrKross > sqrEpsilon * sqrLen0 * sqrLenE) {
            // lines are different
            return 0;
        }

            // lines are the same
            return 2;
        }
```

求解两条线段相交的伪码如下。如果不相交，则函数的返回值为 0；如果只有一个交点，则返回值为 1；如果两条线段重叠并且交集是线段本身，则返回值为 2。返回值是返回的数组I[2]的有效元素数目。该函数与前面介绍的函数一样使用了相对误差检测。

```
int FindIntersection(Point P0, Point D0, Point P1, Point D1, Point2 I[2])
{
    // segments P0 + s * D0 for s in [0, 1], P1 + t * D1 for t in [0,1]

    Point E = P1 - P0;
    float kross = D0.x * D1.y - D0.y * D1.x;
    float sqrKross = kross * kross;
    float sqrLen0 = D0.x * D0.x + D0.y * D0.y;
    float sqrLen1 = D1.x * D1.x + D1.y * D1.y;
    if (sqrKross > sqrEpsilon * sqrLen0 * sqrLen1) {
        // lines of the segments are not parallel
        float s = (E.x * D1.y - E.y * D1.x) / kross;
        if (s < 0 or s > 1) {
            // intersection of lines is not a point on segment P0 + s * D0
            return 0;
        }

        float t = (E.x * D0.y - E.y * D0.x) / kross;
        if (t < 0 or t > 1) {
            // intersection of lines is not a point on segment P1 + t * D1
            return 0;
        }

        // intersection of lines is a point on each segment
        I[0] = P0 + s * D0;
        return 1;
    }

    // lines of the segments are parallel
    float sqrLenE = E.x * E.x + E.y * E.y;
    kross = E.x * D0.y - E.y * D0.x;
    sqrKross = kross * kross;
    if (sqrKross > sqrEpsilon * sqrLen0 * sqrLenE) {

        // lines of the segments are different
        return 0;
    }
```

```
        // Lines of the segments are the same.  Need to test for overlap of
        // segments.
        float s0 = Dot(D0, E) / sqrLen0, s1 = s0 + Dot(D0, D1) / sqrLen0, w[2];
        float smin = min(s0, s1), smax = max(s0, s1);
        int imax = FindIntersection(0.0, 1.0, smin, smax, w);
        for (i = 0; i < imax; i++)
            I[i] = P0 + w[i] * D0;
        return imax;
    }
```

下面的函数用于计算两个区间$[u_0, u_1]$和$[v_0, v_1]$的相交，其中$u_0 < u_1$，$v_0 < v_1$。如果不相交，则函数的返回值为0；如果只有一个交点，则返回值为1，此时w[0]包含了该点；如果它们相交于一个区间，则返回值为2，区间的端点保存在w[0]和w[1]中。

```
int FindIntersection(float u0, float u1, float v0, float v1, float w[2])
{
    if (u1 < v0 || u0 > v1)
        return 0;

    if (u1 > v0) {
        if (u0 < v1) {
            if (u0 < v0) w[0] = v0; else w[0] = u0;
            if (u1 > v1) w[1] = v1; else w[1] = u1;
            return 2;
        } else {
            // u0 == v1
            w[0] = u0;
            return 1;
        }
    } else {
        // u1 == v0
        w[0] = u1;
        return 1;
    }
}
```

7.2　线形对象与折线的相交

计算线形对象与折线相交的最简单的算法，是对折线的每一条线段重复进行线形对象与线段的相交检测。如果应用程序仅需要确定线形对象与折线是否相交，而不需要找到交点，那么，一旦发现折线的一条线段与线形对象相交，算法就可提前结束。

如果折线实际上是一个多边形，并且几何操作将这个多边形作为一个实心体来处理，那么对多边形的每一条边重复使用一条直线或射线的相交检测就足以确定相交与否。如果线形对象本身就是一条线段，那么重复检测是不够的。其中的问题是，线段可能完全包含在多边形内。需要进行额外的检测，特别是对线段的两个端点进行点在多边形内的检测。如果任何一个端点在多边形内，那么线段与多边形相交。如果两个端点都在多边形外，那么对多边形的每一条边重复进行线段与线段的相交检测。

如果经常需要知道一条折线与多条线形对象的相交情况，那么进行一些预处理有助于减少用于穷举各线段搜索的计算时间。一种进行预处理的算法用到了 13.1 节介绍的二维空间分区二叉（BSP）树。该节提供了一些介绍一条线段与一个已经表示为一棵 BSP 树的多边形的相交检测的材料。对 n 条边的穷举搜索是一种$O(n)$级操作。搜索一棵 BSP 树是一种$O(\log n)$级操作。直观上来说，如果线段位于对应于多边形一条边的分区直线的一边，那么不需要进行该线段与位于该分区的另一边的多边形边的相交检测。当然，需要花费进行预处理所需的$O(n \log n)$级时间消耗以建立这棵树。

另一种减少时间消耗的方法是，尽量快速地排除折线的一些线段，使它们不参与到与线形对象的相交检测。6.7 节中的排除方法可用于这一目的。

7.3 线形对象与二次曲线的相交

在本节中，我们将讨论如何检测或寻找线形对象与二次曲线的交点。首先介绍计算线形对象与隐式定义的二次曲线相交的方法。然后再介绍线形对象与圆或弧相交的特殊情形。

7.3.1 线形对象与一般二次曲线的相交

二次曲线用方程$X^{\mathrm{T}}\mathrm{A}X+B^{\mathrm{T}}X+c=0$来隐式表示，其中A为 2×2 的对称矩阵，$B$ 为 2×1 的向量，c 为常数，X为表示曲线上的点的 2×1 的向量。直线$X(t)=P+t\vec{d}$（$t\in\mathbb{R}$）与一条二次曲线的相交可通过如下方法来计算：将直线的方程代入曲线方程中，可得到

$$
\begin{aligned}
0 &= X(t)^{\mathrm{T}}\mathrm{A}X(t)+B^{\mathrm{T}}X(t)+c \\
&= \left(\vec{d}^{\mathrm{T}}\mathrm{A}\vec{d}\right)t^2+\vec{d}^{\mathrm{T}}(2\mathrm{A}P+B)t+\left(P^{\mathrm{T}}\mathrm{A}P+B^{\mathrm{T}}P+c\right) \\
&=: e_2t^2+e_1t+e_0
\end{aligned}
$$

上述二次方程可用二次公式来求解，但必须注意数值问题。例如，当 e_2 接近零，或者当判别式$e_1^2-4e_0e_2$ 接近零时的数值问题。如果该方程有两个不同的实数解，那么直线与曲线相交于两个点。每一个根$\bar{t}$用于计算实际的交点$X(\bar{t})=P+\bar{t}\vec{d}$。如果方程有一个重复的实数解，那么直线与曲线相交于一个点，并与曲线相切。如果方程没有实数解，那么直线与曲线不相交。

如果线形对象是由$t\geqslant 0$所定义的射线，则必须进行额外检测以确定二次方程的根$\bar{t}$是否为非负的。即使包含该射线的直线与曲线相交，射线本身也可能与曲线不相交。类似地，如果线形对象是$t\in[0,1]$所定义的线段，则必须进行额外的检测以确定二次方程的根$\bar{t}$是否也位于区间[0, 1]内。

如果应用程序的目的仅仅是确定线形对象是否与二次曲线相交，并不关心相交于何处，那么可以略过寻找根的计算$q(t)=e_2t^2+e_1t+e_0=0$，以避免二次公式中费时的平方根和除法运算。我们仅需知道$q(t)$是否在$\mathbb{R}$（对直线而言）、[0，∞]（对射线而言）、[0, 1]（对线段而言）上有一个实数根。这可通过 A.5 节中介绍的斯特姆序列来实现。该方法仅使用浮点数加法、减法和乘法来计算 $q(t)$在指定区间内的实数根的数量。

7.3.2 线形对象与圆形曲线的相交

二维空间中的圆可以表示为$\|X-C\|^2=r^2$，其中C为圆心，$r>0$为圆的半径。圆的参数表示为$X(\theta)=C+r\hat{u}(\theta)$，其中$\hat{u}(\theta)=(\cos\theta,\sin\theta)$，$\theta\in[0,2\pi)$。可以用同样的方法来参数表示弧，不同的是$\theta\in[\theta_0,\theta_1]$，$\theta_0\in[0,2\pi),\theta_0<\theta_1$，且$\theta_1-\theta_0<2\pi$。也可以用圆心$C$，半径$r$，以及分别对应于角度$\theta_0$和$\theta_1$的点$A$和$B$来表示。术语圆形对象用来特指圆或弧。

首先考虑一条参数形式的直线$X(t)=P+t\vec{d}$与一个圆$\|X-C\|^2=r^2$的相交。将直线方程代入圆的方程，定义$\vec{\Delta}=P-C$，并得到如下关于t的二次方程：

$$\|\vec{d}\|^2t^2+2\vec{d}\cdot\vec{\Delta}t+\|\vec{\Delta}\|^2-r^2=0$$

方程的根为

$$t=\frac{-\vec{d}\cdot\vec{\Delta}\pm\sqrt{(\vec{d}\cdot\vec{\Delta})^2-\|\vec{d}\|^2(\|\vec{\Delta}\|^2-r^2)}}{\|\vec{d}\|^2}$$

定义$\delta=(\vec{d}\cdot\vec{\Delta})^2-\|\vec{d}\|^2(\|\vec{\Delta}\|^2-r^2)$。如果$\delta<0$，那么直线与圆不相交；如果$\delta=0$，那么直线与圆相切于一个点；如果$\delta>0$，那么直线与圆相交于两个点。

如果线形对象是一条射线，而且$\bar{t}$是二次方程的一个实数根，那么，如果$\bar{t}\geqslant0$，则直线与圆的交点就是射线与圆的交点。类似地，如果线形对象是一条线段，那么，如果$\bar{t}\in[0,1]$，则直线与圆的交点就是线段与圆的交点。

如果圆形对象是一条弧，那么必须检测线形对象与圆的交点是否在弧上。设弧具有端点A和B，弧就是圆上逆时针方向从A到B的部分。注意，从A到B的直线将弧从圆的其余部分分开。图7.1说明了这一点。如果P是圆上的一个点，当且仅当它与弧位于该直线的同一边时，它才在弧上。圆上的点P在弧上的代数条件是$\text{Kross}(P-A,B-A)\geqslant0$，其中$\text{Kross}((x_0,y_0),(x_1,y_1))=x_0y_1-x_1y_0$。

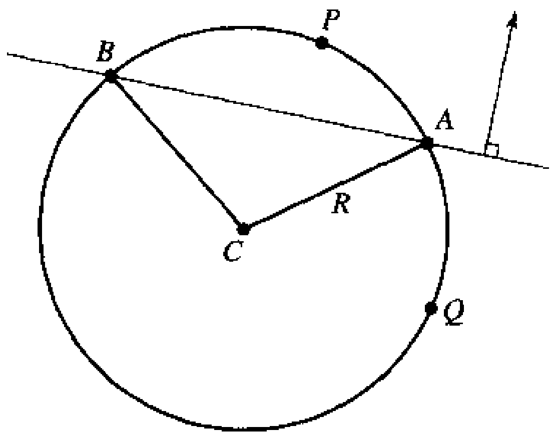


图7.1　圆沿逆时针方向从A到B形成的一段弧。包含A和B的直线将圆分为两部分，即该弧和圆的剩余部分。点P在该弧上，因为它与弧位于直线的同一边。点Q不在弧上，因为它位于直线的另一边

7.4 线形对象与多项式曲线的相交

考虑一条直线$P+t\vec{d}$（$t\in\mathbb{R}$）和一条多项式曲线$X(s)=\sum_{i=0}^{n}\vec{A}_is^i$（$\vec{A}_n\neq\vec{0}$）。设参数

的定义域为 $[s_{min}, s_{max}]$。本节从代数和几何的观点讨论如何计算直线与曲线的交点。

7.4.1　代数方法

如果直线与曲线的交点存在，就可以建立方程 $X(s) = P + t\vec{d}$，并通过用 Kross 算子消除 t 项来求解 s。

$$\sum_{i=0}^{n}\left(\text{Kross}(\vec{d}, \vec{A}_i)\right) s^i = \text{Kross}(\vec{d}, X(s)) = \text{Kross}(\vec{d}, P + t\vec{d}) = \text{Kross}(\vec{d}, P)$$

设 $c_0 = \text{Kross}(\vec{d}, \vec{A}_0 - P)$ 且 $c_i = \text{Kross}(\vec{d}, \vec{A}_i)$（$i \geqslant 1$），上述方程可以改写为多项式方程 $q(s) = \sum_{i=0}^{n} c_i s^i = 0$。可以用一种数值根求解方法来解上述方程，但是应该注意，当 $\vec{d}$ 与 $\vec{A}_n$ 平行（或接近于平行）时，c_n 将为零（或接近于零）。必须检测任何满足 $q(\bar{s}) = 0$ 的 $\bar{s}$ 是否在参数的定义域 $[s_{min}, s_{max}]$ 之内。如果是，则找到了一个交点。

【实例】 设直线为（0，1/2）+ t（2，−1）且多项式曲线为 $X(s) = (0,0) + s(1,2) + s^2(0,-3) + s^3(0,1)$（$s \in [0,1]$）。曲线为单峰的，其 x 值的取值范围为［0，1］，y 值的取值范围为［0，3/8］。多项式方程为 $q(s) = 2s^3 - 6s^2 + 5s - 1 = 0$。其根为 $s = 1, 1 \pm \sqrt{2}/2$。只有根 1 和 $1 - \sqrt{2}/2$ 位于［0，1］内。图 7.2 显示了直线、曲线和交点 $\vec{I}_0$ 和 $\vec{I}_1$。∎

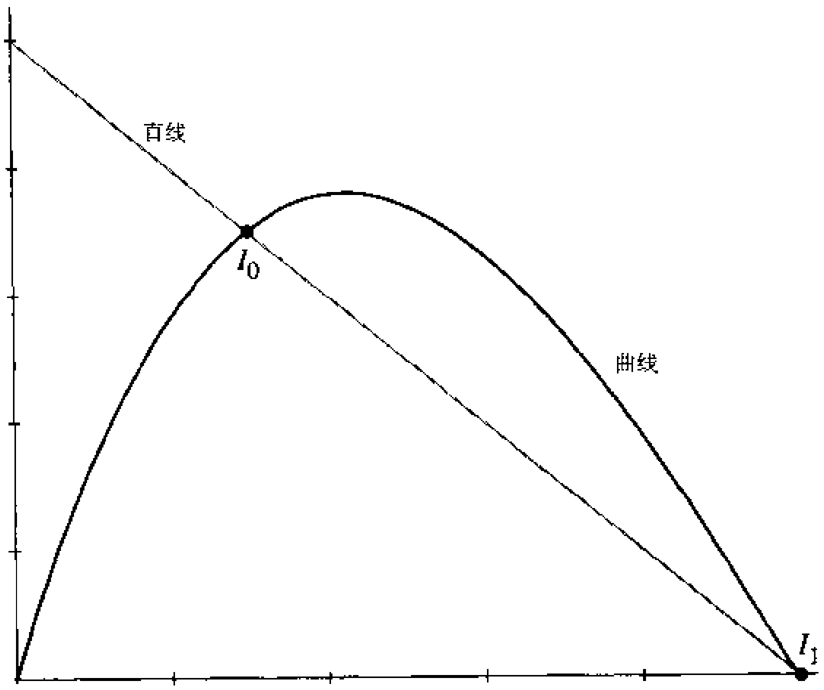


图 7.2　直线与三次曲线的相交

存在多样性或 $q(s)$ 没有大的导数值时，数值根求解方法在求根时可能会出现问题。在几何上，当直线与其中一个交点的切线之间的角度接近于零时，就会出现这种情况。

正如在计算线形对象与二次曲线的相交中出现的问题一样，如果应用程序的目的仅仅是确定线形对象是否与多项式曲线相交，而不关心相交于何处，那么可以略过求 $q(s)=0$ 的根的过程，而使用斯特姆序列（A.5 节）来计算在曲线 $X(s)$ 的定义域 $[s_{min}, s_{max}]$ 内的实数根的数量。如果数量为零，那么直线与多项式曲线不相交。

【实例】使用上例中的数据，我们仅需知道$q(s)=0$ 在 [0，1] 上的实数根的数量。斯特姆序列为$q_0(s)=2s^3-6s^2+5s-1, q_1(s)=6s^2-12s+5, q_2(s)=2(s-1)/3$ 和$q_3(s)=1$。我们有$q_0(0)=-1$，$q_1(0)=5$，$q_2(0)=-2/3$和$q_3(0)=1$，总共有三个符号变化的根。我们还有$q_0(1)=0$，$q_1(1)=-1$，$q_2(1)=0$ 和$q_3(1)=1$，总共有一个符号变化的根。其中符号变化次数之差为 2，因此 $q(s)=0$ 在 [0，1] 上有两个实数根，这说明直线与曲线相交。■

7.4.2 折线逼近

代数方法的求根过程可能需要消耗很多的计算时间。可以减少时间消耗的一种方式是用一条折线来逼近曲线，并寻找直线与该折线的相交。可通过细分（A.8 节）来获得曲线折线。可以利用本章前面介绍的直线—折线检测方法。如果应用程序可以接受这条折线是曲线的一种合适的近似，那么找到的任何相交都可作为直线—曲线相交的近似来使用。然而，得到的交点仅可用来定位曲线上的确切交点。例如，如果一个直线—折线相交出现在线段$(\boldsymbol{X}(s_i),\ \boldsymbol{X}(s_{i+1}))$上，那么，下一步就可以在区间$[s_i, s_{i+1}]$上搜索$q(s)=0$的根。

7.4.3 分级包围

前面提到的代数方法经常需要消耗时间来寻找多项式的根。据推测，最坏的情形是，在花费了计算机的时间来寻找 $q(s)=0$ 的实数根后，根却不存在，即直线与多项式曲线不相交。应用程序可能希望减少确定不相交所需花费的时间，这可以通过提供高层检测以提前知道相交结果来实现。可能更重要的是，如果应用程序执行大量的涉及不同的直线与同一条曲线的直线—曲线相交检测，那么就可以避免用于寻找多项式根的所有时间花费。一些类型的曲线预处理自然也有助于减小时间开销。

一次粗略的检测需要维护曲线的外接多边形。特别地，如果曲线是基于控制点来构建的，并且曲线位于控制点的凸包内，那么可以首先进行直线与凸包（一个凸多边形）的相交检测。如果它们不相交，那么直线与曲线也不相交。如果直线与多边形相交，那么应用程序将进行更费时的直线—曲线相交检测。

另一种方法是利用曲线的轴对齐外接矩形。相对而言，直线—矩形的相交检测所需花费的时间较少，我们将在 7.7 节讨论轴分离时介绍这种方法。如果应用程序为了更早地排除不相交的检测而允许进行更多的循环，那么可以得到这种算法的一种变体，即建立不同级的外接矩形，每一级都提供比上一级（在感觉上）更好的适应。而且，如果直线在某些级上与外接矩形不相交，那么在比这一矩形结点更高级的矩形上不存在需要处理的点，这是因为直线不能仅在一个局部范围内与曲线相交。图 7.3 说明了这一点。图中显示了具有两级的一条曲线。直线与顶级矩形相交，因此必须分析其下一级的相交情况。直线与下一级的左边子矩形相交，因此进一步的相交检测需要进行直线—矩形或直线—曲线检测。直线与下一级的右边子矩形不相交，因此，直线不可能与包含于该矩形的曲线相交。不需要对该级的子树进行进一步的处理。

其中主要的问题自然就是如何构建曲线的轴对齐外接矩形。对于特殊类型的曲线，特别是贝塞尔曲线，这并不难。曲线的轴对齐外接矩形一般并不是相对曲线的控制点而构建的具有最小面积的矩形。矩形的级可通过分解曲线并使矩形适应相对于每一个子曲线的控

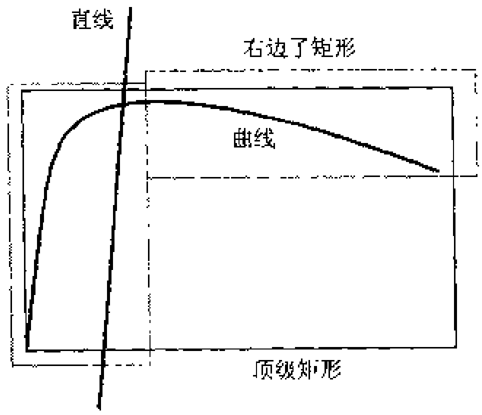


图 7.3　用分级外接界框检测直线－曲线的相交

制点来构建。对于一般的多项式曲线，寻找最小面积的轴对齐外接矩形似乎与寻找直线与曲线相交的算法一样复杂。矩形在 x 方向上的扩展由曲线上的 x 极端点确定。x 极端点的基本特征是，在该点上具有曲线的垂直切线。在数学上，如果 $x'(t)=0$，那么在$(x(t), y(t))$处具有垂直切线。类似地，y 极端点的基本特征是，在该点上具有曲线的水平切线，此时 $y'(t)=0$。每一个导数方程都是一个可求数值解的多项式方程，但是这样做将不能实现利用外接矩形来避免在一个直线与曲线不相交的区域内进行费时的求根运算的目标。如果原曲线是三次曲线，那么其导数方程可利用二次公式来求解，这将不会有问题。如果应用程序需要进行多条直线与一条曲线的相交检测，这也不会有问题。与进行直线－曲线相交检测时进行求根所需的所有时间花费相比，计算外接矩形的预处理时间是微不足道的。

7.4.4　单调分解

现在假设对任意的$t \in [t_{\min}, t_{\max}]$，$x'(t) \neq 0$。曲线在 x 处是单调的，不是严格递增就是严格递减。在这种特殊情形中，轴对齐的外接矩形沿 x 轴的扩展对应于$x(t_{\min})$ 和$x(t_{\max})$。相似的讨论适用于在曲线定义域上$y'(t) \neq 0$时的情形。一般地，如果对于$t \in [a, b] \subseteq [t_{\min}, t_{\max}]$，有$x'(t) \neq 0$且$y'(t) \neq 0$，那么曲线段是单调的且轴对齐外接矩形由点$(x(a), y(a))$和$(x(b), y(b))$确定。确定一个导数方程是否有根是斯特姆次序的一种应用。

现在我们的方法是在要寻找单调曲线段的参数区间上使用简单的对分法来分解曲线。如果$[a, b]$是对分范围的一个子区间，曲线在该区间上是单调的，那么不需进行进一步的分解，轴对齐的外接矩形就是曲线段的外接矩形。在理论上，经过几级对分后，我们将获得少量的单调曲线段和与它们对应的外接矩形。可对直线进行直线－矩形相交检测，以选出与直线不相交的单调曲线段。然而，如果外接矩形的数量很大，就总能将原来的矩形作为树的叶子节点，并一次组合一些矩形来建立父节点，以从底向上建立不同的级别。通过计算，父节点的外接矩形包含其子节点的外接矩形。父节点也可组合，这种过程最终将生成树的根节点，它仅有一个外接矩形。图 7.3 中说明的相交方法可用来处理这棵树。

一种递归的分解可用来寻找单调曲线段。应该慎重选择递归要求的停止条件。如果导数方程$x'(t)=0$且$y'(t)=0$在当前区间内都具有零根，那么曲线在该区间内是单调的，同时递归终止。如果其中一个方程在当前区间内仅有惟一的一个根，而另一个方法没有根，就需要进行分解。用于分解的 t 值也可能就是根，此时两个子区间都将报告存在一个根，虽

然实际上只有一个根。例如，考虑$(x(t), y(t)) = (t, t^2)$（$t \in [-1, 1]$）。导数方程$x'(t) = 0$没有根，因为对所有的t，有$x'(t) = 1$。但是$y'(t) = 2t = 0$在该区间内有一个根。细分值为$t = 0$。方程$y'(t) = 0$在子区间［-1，0］上有一个根，并且在子区间［0，1］上也有一个根，然而它们其实是同一个根。应该检查子区间的终点，以避免进行不必要的递归。典型的情形是子区间的根一般出现在区间的内点。一旦找到有惟一内点根的子区间，就可以应用健壮的二分法来寻找根并在根出现的地方分解子区间，并终止对子区间的递归。

在进行多条直线与一条曲线的相交检测的应用程序中，可以利用数值根的求解方法来求解$x'(t) = 0$且$y'(t) = 0$，并将寻找单调曲线段作为一种预处理。

7.4.5 栅格方法

可以使用一种栅格方法，虽然这种方法可能是非常费时的。先建立包含曲线的$[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$。然后建立一个$N \times M$栅格，用来表示该矩形区域。均匀地选取栅格点为$(x_i, y_j)$，其中$0 \leqslant i < N$且$0 \leqslant j < M$。即$x_i = x_{\min} + (x_{\max} - x_{\min})i/(N-1)$和$y_j = y_{\min} + (y_{\max} - y_{\min})j/(M-1)$。直线和曲线都画在该格子内。应该将曲线参数的步长大小选择为足够小，从而可以对曲线采样得到相邻的栅格值，但由于可能其栅格单元由于多个曲线样本落在同一单元，其栅格单元因而可能被画很多次。通过基于弧长而不是曲线参数来对曲线采样，可以减少这种重复画的次数。如果栅格被初始化为 0，那么用像素与 1 的或来画直线，并用像素与 2 的或来画曲线。最终值为 3 的像素就是与直线－曲线相交相关的像素。图 7.4 说明了这一点。

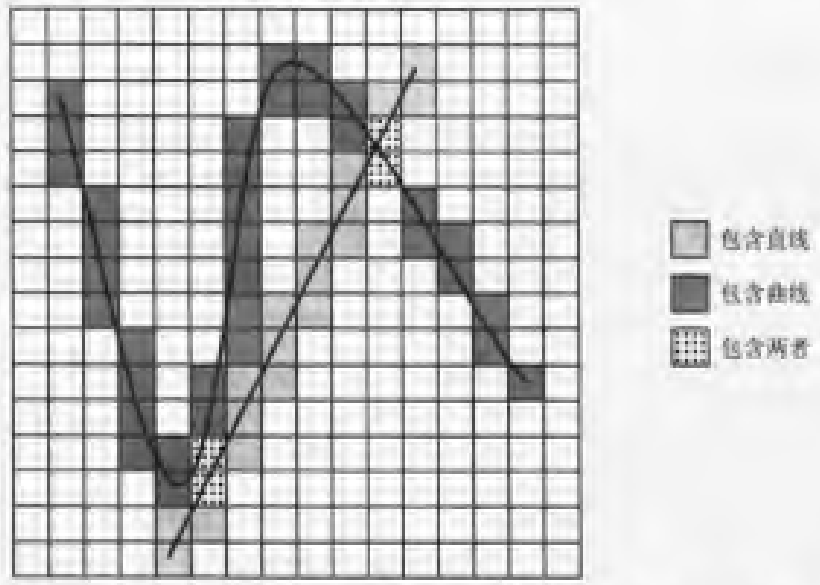


图 7.4　在起始于零的格子中被栅格化的一条直线和一条曲线。直线使用掩码为 1（浅灰色）的或运算被栅格化，曲线使用掩码为 2（深灰色）的或运算被栅格化。同时容纳直线和曲线的格子单元的值为 3（点状）

这种方法的效率取决于格子的大小是如何选取的。如果它被选得太大，那么即使直线和曲线不相交，它们也将通过同一个像素。这时栅格方法将把不相交报告为相交。图 7.4 中左下部分的格子显示了这种情况。如果格子选得太小，那么将花费许多时间来栅格化曲

线，特别是花费在直线与曲线不相交的像素上。

正如用于折线的方法一样，应用程序可以选择接受像素值作为实际的直线－曲线相交的近似。如果想要更高的精度，那么标记为 3 的像素（而且也可能是直接相邻的像素）可用来定位相交的位置。如果邻近的一块像素被标记，如图 7.4 的右上方格子所示，并且如果因为具有惟一的交点，应用程序选择认为相交出现在这块像素内，那么近似的合理选择就是像素位置的平均值。如果应用程序选择不接受像素值作为近似值，那么可以将原直线和曲线出现在像素内的样本的参数与该像素一起存储。这些参数可以用于利用数值根解法或数值距离计算法的相交搜索。

7.5　二次曲线之间的相交

我们将列出计算两条用二次方程隐式定义的曲线相交的一般代数方法。还会介绍圆形对象的特殊情形，说明如何计算圆弧之间的相交。同时还将介绍计算两个椭圆的交点的不同方法，以说明如何处理两条曲线之间的更一般的相交问题，每一种方法都对特定的函数定义了一条阶层曲线。

7.5.1　一般二次曲线之间的相交

给定由二次方程 $F(x,y)=\alpha_{00}+\alpha_{10}x+\alpha_{01}y+\alpha_{20}x^2+\alpha_{11}xy+\alpha_{02}y^2=0$ 和 $G(x,y)=\beta_{00}+\beta_{10}x+\beta_{01}y+\beta_{20}x^2+\beta_{11}xy+\beta_{02}y^2=0$ 隐含定义的两条曲线，它们之间的交点可通过消去 y 并得到一个四次多项式方程 $H(x)=0$ 来计算。在消元的过程中，y 通过多项式方程 $y=R(x)$ 与 x 联系在一起。$H(x)=0$ 的每一个根 $\bar{x}$ 都用于计算 $\bar{y}=R(\bar{x})$。数对 $(\bar{x},\bar{y})$ 就是两条曲线的交点。

【实例】方程 $x^2+6y^2-1=0$ 定义了一个中心位于原点的椭圆。方程 $2(x-2y)^2-(x+y)=0$ 确定了一条顶点位于原点的抛物线。图 7.5 显示了这两条曲线的图形。椭圆方程可以改写为 $y^2=(1-x^2)/6$。将其代入抛物线方程，可得

$$
\begin{aligned}
0&=2x^2-8xy+8y^2-x-y=2x^2-8xy+8(1-x^2)/6-x-y\\
&=-(8x+1)y+(2x^2-3x+4)/3
\end{aligned}
$$

其解为

$$
y=\frac{2x^2-3x+4}{3(8x+1)}=:R(x)
$$

将其代入椭圆方程，可得

$$
\begin{aligned}
0&=x^2+6y^2-1\\
&=x^2+6\left(\frac{2x^2-3x+4}{3(8x+1)}\right)^2-1\\
&=\frac{9(8x+1)^2(x^2-1)+6(2x^2-3x+4)^2}{9(8x+1)^2}\\
&=\frac{200x^4+24x^3-139x^2-96x+29}{3(8x+1)^2}
\end{aligned}
$$

因此，必有

$$0 = 200x^4 + 24x^3 - 139x^2 - 96x + 29 =: H(x) \quad \blacksquare$$

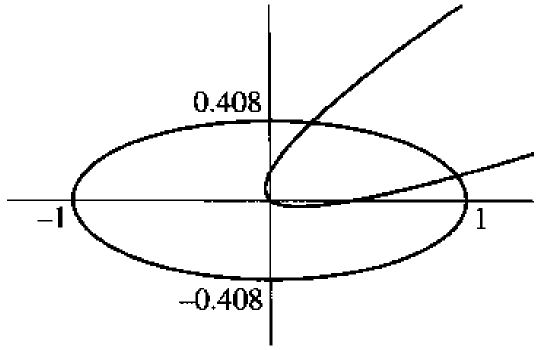


图 7.5　椭圆和双曲线相交

多项式方程$H(x)=0$ 有两个实数根 $x_0 \doteq 0.232856$ 和$x_1 \doteq 0.960387$。将它们代入关于 y 的有理多项式，可得$y_0 = R(x_0) \doteq 0.397026$ 和$y_1 = R(x_1) \doteq 0.113766$。点$(x_0, y_0)$和$(x_1, y_1)$就是椭圆和抛物线的交点。

多项式方程的一般解法将在 A.2 节中进行详细讨论。

当根的数目具有偶数多样性或导数函数在根附近的值特别小时，任何求根的方法都可能引起数值问题。当两条曲线有一个交点且在交点处的曲线与切线之间的角度接近于零时，这类问题将上升为几何问题。如果需要特别高的精度且不想漏掉交点，就需要保证求根的方法在进行一些特殊的计算时特别健壮。

如果应用程序仅仅需要知道曲线是否相交，而不需要知道相交于何处，那么求根的斯特姆次序可用于求解$H(x)=0$。

7.5.2　圆形二次曲线之间的相交

设两个圆表示为$\|X - C_i\|^2 = r_i^2$，其中 $i=0, 1$。如果它们之间存在交点，那么交点可通过如下的方法来确定。定义$\vec{u} = C_1 - C_0 = (u_0, u_1)$。定义$\vec{v} = (u_1, -u_0)$。注意$\|\vec{u}\|^2 = \|\vec{v}\|^2 = \|C_1 - C_0\|^2$和$\vec{u} \cdot \vec{v} = 0$ 。交点可表示为如下的形式

$$X = C_0 + s\vec{u} + t\vec{v} \tag{7.1}$$

或

$$X = C_1 + (s-1)\vec{u} + t\vec{v} \tag{7.2}$$

其中s 和t 用如下的参数来建立。将方程（7.1）代入$\|X - C_0\|^2 = r_0^2$，可得

$$(s^2 + t^2)\|\vec{u}\|^2 = r_0^2 \tag{7.3}$$

将方程（7.2）代入$\|X - C_1\|^2 = r_1^2$，可得

$$((s-1)^2 + t^2)\|\vec{u}\|^2 = r_1^2 \tag{7.4}$$

将方程（7.3）和方程（7.4）代入，并求解s，可得

$$s = \frac{1}{2}\left(\frac{r_0^2 - r_1^2}{\|\vec{u}\|^2} + 1\right) \tag{7.5}$$

将方程（7.5）代入方程（7.3），并求解t^2，可得

$$t^2=\frac{r_0^2}{\|\vec{u}\|^2}-s^2=\frac{r_0^2}{\|\vec{u}\|^2}-\frac{1}{4}\left(\frac{r_0^2-r_1^2}{\|\vec{u}\|^2}+1\right)^2 \tag{7.6}$$

$$=-\frac{(\|\vec{u}\|^2-(r_0+r_1)^2)(\|\vec{u}\|^2-(r_0-r_1)^2)}{4\|\vec{u}\|^4}$$

要使这些方程有解，方程（7.6）的右边必须是非负的。因此，分子为负数：

$$(\|\vec{u}\|^2-(r_0+r_1)^2)(\|\vec{u}\|^2-(r_0-r_1)^2)\leqslant 0 \tag{7.7}$$

如果$w=\|\vec{u}\|$，不等式（7.7）的左边定义了一个关于w的二次函数，其图形为开放向上的抛物线。其根为$w=|r_0-r_1|$和$w=|r_0+r_1|$。对于负的二次函数，只允许w的值在两个根之间。不等式（7.7）因此等价于

$$|r_0-r_1|\leqslant\|\vec{u}\|\leqslant|r_0+r_1| \tag{7.8}$$

如果$\|\vec{u}\|=|r_0+r_1|$，每一个圆都位于另一个的外面，但是刚好相切。它们的交点为$C_0+(r_0/(r_0+r_1))\vec{u}$。如果$\|\vec{u}\|=|r_0-r_1|$，圆相套，但刚好相切。如果$\|\vec{u}\|=0$且$r_0=r_1$，那么它们是同一个圆。否则，交点为$C_0+(r_0/(r_0-r_1))\vec{u}$。如果$|r_0-r_1|<\|\vec{u}\|<|r_0+r_1|$，那么这两个圆相交于两个点。从方程（7.5）求得的s值和从方程（7.6）中求得的值的平方根t值可以用来计算交点，即$C_0+s\vec{u}+t\vec{v}$。图 7.6 显示了两个圆之间的不同关系。

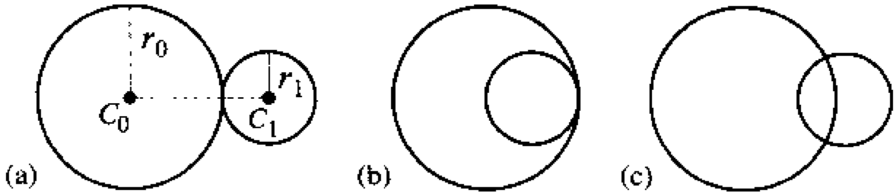


图 7.6 两个圆的关系，$\vec{u}=C_1-C_0$: (a) $\|\vec{u}\|=|r_0+r_1|$; (b) $\|\vec{u}\|=|r_0-r_1|$; (c) $|r_0-r_1|<\|\vec{u}\|<|r_0+r_1|$

如果两个圆形对象中至少有一个是弧，那么必须利用本章前面介绍的判断圆上的点是否在弧上的算法，来检测两个圆的交点是否在弧上。

7.5.3 椭圆之间的相交

前面讨论的检测或寻找交点的代数方法当然也适用于椭圆，因为椭圆可用二次方程隐式地定义。在某些应用程序中，需要知道更多的信息，而不仅仅需要知道交点。特别地，如果椭圆作为外接区域，那么知道一个椭圆是否完全包含另一个椭圆是非常重要的。用于定义两个椭圆的两个二次方程的代数方法并未提供这种信息。对椭圆$\mathcal{E}_0$和$\mathcal{E}_1$的更精确的查询包括：

- $\mathcal{E}_0$和$\mathcal{E}_1$是否相交？
- $\mathcal{E}_0$和$\mathcal{E}_1$是否分离？即是否存在一条直线，使得这两个椭圆分别位于该线的两边？
- $\mathcal{E}_0$是否完全包括$\mathcal{E}_1$，或者$\mathcal{E}_1$是否完全包括$\mathcal{E}_0$？

设椭圆$\mathcal{E}_i$由二次方程$Q_i(X)=X^T\mathrm{A}_iX+B_i^TX+c_i$（$i=0,1$）定义。假设$\mathrm{A}_i$为正无穷大。

在这种情形下，$Q_i(X)<0$定义了椭圆的内部，而$Q_i(X)>0$定义了椭圆的外部。

这里的讨论集中于二次函数的阶层曲线。A.9.1 节讨论了函数的阶层集。所有由$Q_0(x,y)=\lambda$定义的阶层曲线都是椭圆，惟一的例外是最小值（负数）λ，此时方程定义一个点，即每一个椭圆的中心。$Q_1(x,y)=0$定义的椭圆一般与Q_0的许多阶层曲线相交。要解决的问题是寻找在$\mathcal{E}_1$上的任何(x,y)处获得的最小的阶层值λ_0和最大的阶层值λ_1。如果$\lambda_1<0$，那么$\mathcal{E}_1$将包含于$\mathcal{E}_0$内。如果$\lambda_0>0$，那么$\mathcal{E}_1$和$\mathcal{E}_0$是分开的或者$\mathcal{E}_1$包含$\mathcal{E}_0$。否则，$0\in[\lambda_0,\lambda_1]$，并且这两个椭圆相交。图 7.7，图 7.8 和图 7.9 显示了这三种可能性。这些图形显示了一个椭圆$\mathcal{E}_1$和另一个椭圆$\mathcal{E}_0$的阶层曲线之间的关系。

这可以明确地表示为一种强制的优化条件，可以利用拉格朗日乘子法（参见 A.9.3 节）来求解：用限制条件$Q_1(X)=0$来优化$Q_0(X)$。定义$F(X,t)=Q_0(X)+tQ_1(X)$。对X的分量微分可得$\vec{\nabla}F=\vec{\nabla}Q_0+t\vec{\nabla}Q_1$。对$t$微分可得$\partial F/\partial t=Q_1$。设$t$的导数等于零将得到限制条件$Q_1=0$。设$X$的导数等于零，将得到关于某些$t$的$\vec{\nabla}Q_0+t\vec{\nabla}Q_1=\vec{0}$。在几何意义上，这意味着梯度是平行的。

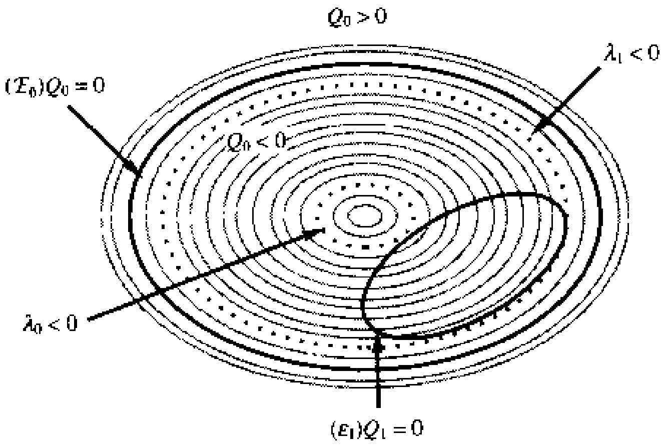


图 7.7　$\mathcal{E}_1$包含在$\mathcal{E}_0$中。$\mathcal{E}_0$对$\mathcal{E}_1$的阶层曲线的最大值λ_1是负数

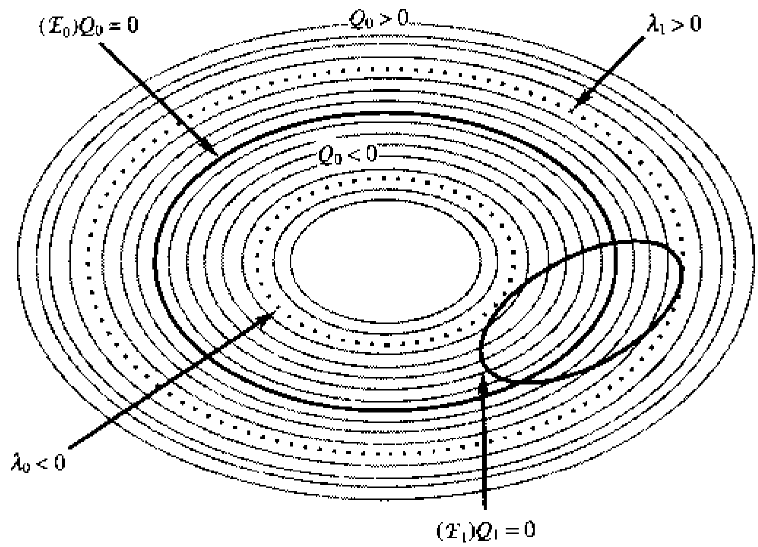


图 7.8　$\mathcal{E}_1$横切地与$\mathcal{E}_0$相交。$\mathcal{E}_0$对$\mathcal{E}_1$的阶层曲线的最小值λ_0是负数，最大值λ_1是正数

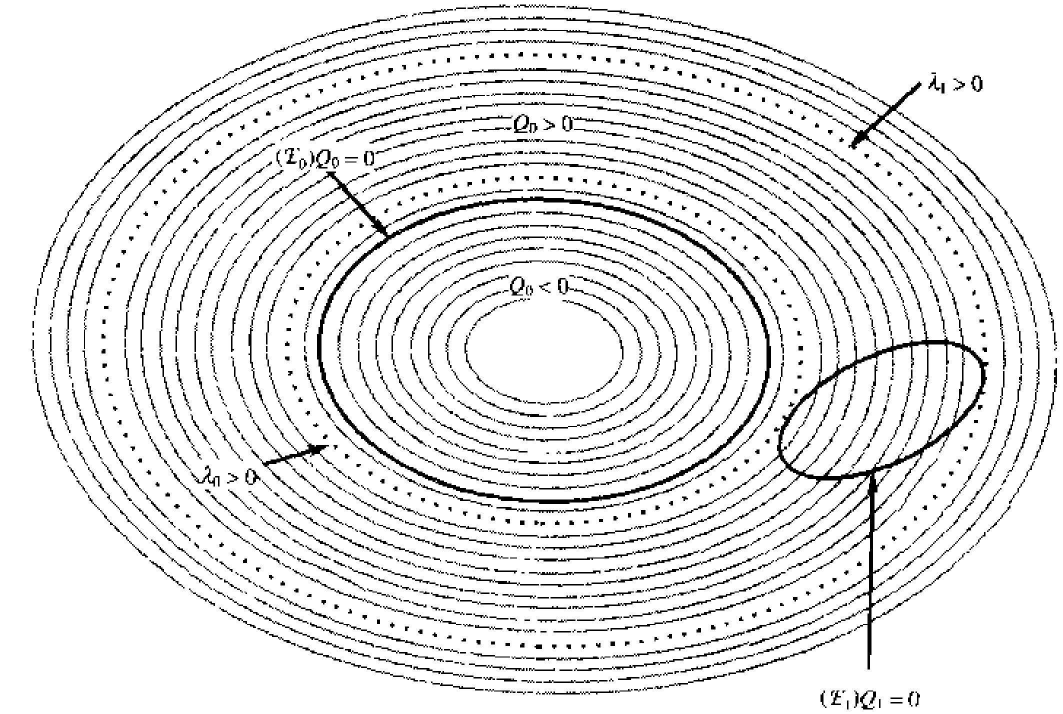


图 7.9　$\mathcal{E}_1$从$\mathcal{E}_0$中分离出来　$\mathcal{E}_0$对$\mathcal{E}_1$的阶层曲线的最小值 λ_0 是正数

由于有$\vec{\nabla}Q_i=2\mathrm{A}_iX+B_i$，因此

$$\vec{0}=\vec{\nabla}Q_0+t\vec{\nabla}Q_1=2(\mathrm{A}_0+t\mathrm{A}_1)X+(B_0+tB_1)$$

对其求解 X 可得

$$X=-\frac{1}{2}(\mathrm{A}_0+t\mathrm{A}_1)^{-1}(B_0+tB_1)=\frac{1}{\delta(t)}\vec{Y}(t)$$

其中$\mathrm{A}_0+t\mathrm{A}_1$是关于 t 的线性多项式的一个矩形，并且其行列式为$\delta(t)$，这是一个二次多项式。$\vec{Y}(t)$的分量是关于 t 的二次多项式。将其代入 $Q_1(X)=0$，可得

$$p(t):=\vec{Y}(t)^{\mathrm{T}}\mathrm{A}_1\vec{Y}(t)+\delta(t)B_1^{\mathrm{T}}\vec{Y}(t)+\delta(t)^2C_1=0 \tag{7.9}$$

这是关于 t 的二次多项式。可以求解其根和对应的 X 值，并评估$Q_0(X)$。最小值和最大值分别存储为 λ_0 和 λ_1，并利用前面已进行的与零的比较结果。

与原来用于寻找交点的代数方法一样，这种方法将得到二次多项式。可是当前的方法将回答关于椭圆之间的相对位置（分开或包含）的问题，而原来的方法并不回答这类问题。

【实例】 考虑 $Q_0(x,y)=x^2+6y^2-1$和 $Q_1(x,y)=52x^2-72xy+73y^2-32x-74y+28$。图 7.10 显示了这两个椭圆的图形。各参数为

$$\mathrm{A}_0=\begin{bmatrix}1&0\\0&6\end{bmatrix},\quad B_0=\begin{bmatrix}0\\0\end{bmatrix},\quad C_0=-1,\quad \mathrm{A}_1=\begin{bmatrix}52&-36\\-36&73\end{bmatrix},$$

$$B_1=\begin{bmatrix}-32\\-74\end{bmatrix},\quad C_1=28$$

由此可导出

$$\vec{Y}(t)=\begin{bmatrix}4t(625t+24)\\t(2500t+37)\end{bmatrix},\quad \delta(t)=2500t^2+385t+6$$

方程（7.9）中的多项式为$p(t) = -156250000t^4 - 48125000t^3 + 1486875t^2 + 94500t + 1008$。它的两个实数根分别为$t_0 \doteq -0.331386$和$t_1 \doteq 0.0589504$。对应的$X(t)$值分别为$X(t_0) = (x_0, y_0) \doteq (1.5869, 1.71472)$和$X(t_1) = (x_1, y_1) \doteq (0.383779, 0.290742)$。轴对齐椭圆在这些点的阶层值分别为$Q_0(x_0, y_0) \doteq -0.345528$和$Q_0(x_1, y_1) = 19.1598$。由于$Q_0(x_0, y_0) < 0 < Q_0(x_1, y_1)$，因此椭圆相交。图 7.10 显示了具有极值 Q_0 的在 $Q_1 = 0$ 处的两个点。■

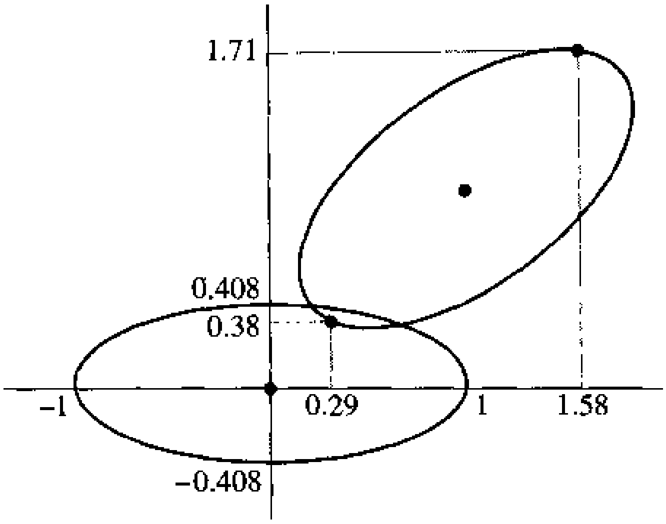


图 7.10　两个椭圆的相交

7.6　多项式曲线之间的相交

考虑两条多项式曲线$X(s) = \sum_{i=0}^{n} \vec{A}_i s^i$，其中$\vec{A}_n \neq \vec{0}$且$s \in [s_{\min}, s_{\max}]$，以及$Y(t) = \sum_{j=0}^{m} \vec{B}_j t^j$，其中$\vec{B}_m \neq \vec{0}$且$t \in [t_{\min}, t_{\max}]$。本节分别介绍如何从代数和几何的观点来计算这样的两条曲线之间的距离。

7.6.1　代数方法

直接的代数方法是建立方程$X(s) = Y(t)$，并求解参数s和t。我们知道，这个向量方程可导出两个次数为$\max\{n, m\}$的含两个未知数s和t的多项式方程。可以利用消元法来得到一个只含一个未知数的方程$q(s) = 0$。求解的方法是在介绍直线与多项式曲线相交一节中介绍的方法的简单扩展，只是$q(s)$的次数更高（对于直线，$m = 1$；对于曲线，一般有$m > 1$）。

7.6.2　折线逼近

用代数方法来求根可能是很费时的。与 7.4 节中的直线－曲线的相交检测相似，通过用折线来逼近曲线并求两条折线的相交，可以极大地降低计算的复杂性。用细分的方法可以得到相关的折线。如果应用程序认可折线是曲线的合理近似，那么折线之间的相交可以看成是曲线之间的相交的近似。然而，折线的交点也可以用做开始搜索精确的交点的初始值。

7.6.3　分级包围

在 7.4 节中，我们讨论了利用外接多边形、外接矩形或分级的外接矩形来进行粗略检

测，以尽早排除不相交的对象。相同的思想可用于曲线–曲线的相交检测。如果曲线由控制点定义，且具有凸包性质，那么应该首先进行可被尽早排除的不相交的检测，以确定凸多边形是否与曲线相交。如果不相交，那么曲线也不相交。如果相交，那么可进行更精细的检测，比如前面提到的代数方法。也可使用利用矩形树的分级方法。每一条曲线都有为它构建的矩形级。为了确定相交检测，必须对从每棵树的矩形中各取一个矩形构成的矩形对并进行比较。这就是 Gottschalk，Lin 和 Manocha（1996）所用的三维空间有向外接矩形法。在二维空间中，该方法可用于曲线段而不是折线。问题之一是需要进行额外的分析来确定箱–箱相交于哪一点的检测，这比用代数方法来进行曲线–曲线的相交检测更加费时。在这一点上，箱–箱相交检测的简易性所带来的好处还不及它所需额外的时间花销所引起的坏处。用于箱–箱相交检测的许多时间开销主要取决于箱所处级别的深度。另一个问题是为曲线建立轴对齐封闭箱的问题。这在 7.4 节中已经讨论过了。

7.6.4　栅格方法

最后，也可以使用一种栅格方法，虽然这种方法可能是非常费时的。建立一个轴对齐封闭箱$[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$，让它包含两条曲线。建立一个$N \times M$的栅格来表示箱的区域。均衡地选取格子点$(x_i, y_j)$，其中 $0 \leqslant i < N$且 $0 \leqslant j < M$。即$x_i = x_{\min} + (x_{\max} - x_{\min})i/(N-1)$和$y_j = y_{\min} + (y_{\max} - y_{\min})j/(M-1)$。每一条曲线都画在栅格内。曲线参数的步长应该选为足够小，这样才能在对曲线采样时产生相邻的栅格值。由于多个曲线样本落入同一单元，因此一个栅格单元可能被画多次。基于弧长而不是基于曲线参数来对曲线采样，可尽可能地减少这种重复被画的现象。如果栅格被初始化为 0，通过像素与 1 的或运算来画第一条曲线，并且通过像素与 2 的或运算来画第二条曲线，那么与曲线–曲线相交相关的像素就是那些最终值为 3 的像素（如图 7.11 所示）。

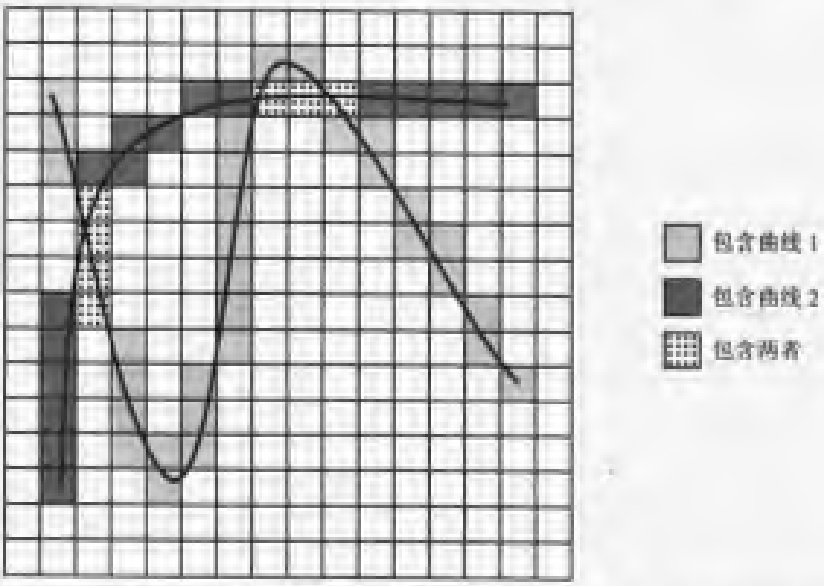


图 7.11　在起始于零的格子中被栅格化的两条曲线。第一条曲线使用掩码为 1（浅灰色）的或运算被栅格化。第二条曲线使用掩码为 2（深灰色）的或运算被栅格化。同时容纳两条曲线的格子单元的值为 3（点状）

注意，最左边的像素块（点状的单元）并不能确切地说明曲线可能相交于何处。一般的问题是两条曲线可能相距很近但不相交，却被栅格化到同一像素内。解决的方法是增加单元格的数目，同时减小单元格的大小，以得到更高分辨率的格子。一般情况下开始时并不知道单元格的大小应该是多少才能适当地检测相交，并且不会产生错误的结果。

与折线方法一样，应用程序能选择接受将像素值作为实际的曲线－曲线相交的近似。如果要求更高的精度，那么标记为 3 的像素（也可能包括其近邻）可作为确定相交出现的位置的一种定位。如果像素的邻近块被标记（如图 7.11 中的左边像素）并且应用程序可以确定该块的出现是由于曲线的单一相交，那么像素位置的平均值就是合理的近似选择。如果应用程序选择不接受像素值作为近似，那么它可以将样本出现在同一像素的原曲线的参数与该像素一起存储。这些参数值可以用来启动一种搜索，以确定用数值根求解方法或数值距离计算方法计算得到的相交。

7.7 轴分离方法

如果给定集合 S 中的任意两个点 P 和 Q，线段 $(1-t)P+tQ$（$t\in[0,1]$）也位于 S 内，那么 S 是凸集。本节描述二维空间内的轴分离方法，这是一种确定两个固定的凸对象是否相交的方法。这种思想扩展到运动的凸对象，对于通过计算第一次接触的时间来预测对象碰撞，以及计算接触集来说是非常有用的。考虑两种类型的集合问题。第一个问题是检测相交的问题，仅说明固定的凸对象是否存在相交，或者运动对象是否会发生相交。第二个问题是寻找相交的问题，这需要计算两个固定的凸对象的交集，或者运动对象第一次相交的交集。本节将描述二维空间中的凸多边形的这两种类型的问题。

我们在本节中采用如下的表示方法。设 C_j（$j=0$，1）是顶点为 $\{V_i^{(j)}\}_{i=0}^{N_j-1}$（按逆时针排序）的凸多边形。多边形的边为 $\vec{e}_i^{\,(j)}=V_{i+1}^{(j)}-V_i^{(j)}$，其中 $0\leqslant i<N_j$，$V_{N_j}^{(j)}=V_0^{(j)}$。边的指向外面的法线向量为 $\vec{d}_i^{(j)}=\operatorname{Perp}\left(\vec{e}_i^{\,(j)}\right)$，其中 $\operatorname{Perp}(x,y)=(y,-x)$。

7.7.1 投影到直线上的分离

两个凸对象不相交的检测可以简单地描述为：如果存在一条直线，两个对象在该直线上的投影区间不相交，那么这两个对象不相交。这样的直线叫做分离线，或者更一般地称为分离轴（参见图 7.12）。分离线的平移还是分离线，因此考虑经过原点的直线就足够了。给定一条经过原点、单位长度方向为 $\vec{d}$ 的直线，一个凸集 C 在该直线上的投影为区间

$$I=[\lambda_{\min}(\vec{d}),\lambda_{\max}(\vec{d})]=[\min\{\vec{d}\cdot\vec{X}:\vec{X}\in C\},\max\{\vec{d}\cdot\vec{X}:\vec{X}\in C\}]$$

其中可能 $\lambda_{\min}(\vec{d})=-\infty$ 或者 $\lambda_{\max}(\vec{d})=+\infty$；当凸集没有边界时，将出现这类情形。如果存在一个方向 $\vec{d}$，两个凸集 C_0 和 C_1 在其上的投影 I_0 和 I_1 不相交，那么它们是分离的。当下列条件满足时，它们是不相交的：

$$\lambda_{\min}^{(0)}(\vec{d})>\lambda_{\max}^{(1)}(\vec{d})\ \text{或}\ \lambda_{\max}^{(0)}(\vec{d})<\lambda_{\min}^{(1)}(\vec{d}) \tag{7.10}$$

式中的上标对应于凸集的索引。虽然上式中假定 $\vec{d}$ 是单位长度的向量，但是改变向量的长度并不会改变比较的结果。这是因为 $\lambda_{\min}(t\vec{d})=t\lambda_{\min}(\vec{d})$ 且 $\lambda_{\max}(t\vec{d})=t\lambda_{\max}(\vec{d})$（其中

$t > 0$)。当用$-\vec{d}$替换式中的$\vec{d}$时，这对比较式的布尔值也保持不变。这是因为$\lambda_{\min}(-\vec{d}) = -\lambda_{\max}(\vec{d})$且$\lambda_{\max}(-\vec{d}) = -\lambda_{\min}(\vec{d})$。当$\vec{d}$不是单位长度向量时，从分离线检测中获得的区间并不是对象在直线上的投影，而是投影区间的缩放版本。我们不区分缩放的投影和正常的投影。我们也使用术语分离方向（separating direction）来描述分离线的方向向量，并不要求这个向量是单位长度的向量。

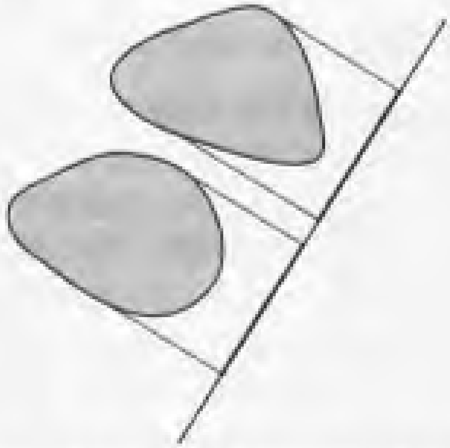


图 7.12　不相交的凸对象和它们的一条分离线

请注意，在二维空间中，术语分离线或分离轴可能使人感到迷惑。分离线分离对象在该直线上的投影。分离线不将平面分离为各包含一个对象的两个区域。在三维空间中，这个术语应该不会使人感到迷惑，因为不需要指定一个平面来将空间划分为两个区域，每一个区域各包含一个对象。一条直线并不能明确地将空间划分为两部分。

7.7.2　固定凸多边形的分离

对于一对凸多边形，进行分离检测时该集合仅包括多边形的边的法线向量。图 7.13（a）显示了被一个多边形的一条边的法线所确定的方向所分开的两个不相交的多边形。图 7.13（b）显示了两个相交的多边形，不存在分离方向。仅需检测一条边的法线，其直观原因是存在仅仅互相接触而不贯通的两个凸多边形。图 7.14 显示了这三种可能的构形：边－边接触，顶点－边接触和顶点－顶点接触。一个对象从另一个对象平移一个微小的距离时出现一条分离线，两个多边形之间的直线垂直于该分离线。

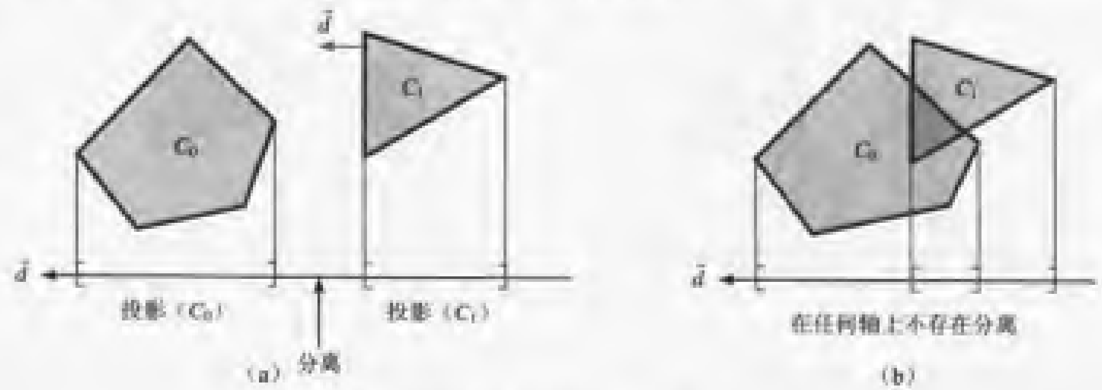


图 7.13　（a）不相交的凸多边形　（b）相交的凸多边形



图 7.14　（a）边－边接触　（b）顶点－边接触　（c）顶点－顶点接触

S 仅包括边法线，其数学证明是基于如下的事实：如果$\vec{d}$是分离方向，但不是任何一个多边形的边法线，那么必定存在一条边法线为分离方向。设$\vec{d}=(\cos\theta,\sin\theta)$为非边法线的分离方向。为了讨论的方便，假设$C_0$在分离线上的投影在$C_1$的投影的左边（如果在右边的话，可以进行相似的讨论）。由于$\vec{d}$不是边法线，因此只有C_0的一个顶点V_0映射为$\lambda_{\max}^{(0)}$，并且只有C_1的一个顶点V_1映射为$\lambda_{\min}^{(1)}$。设θ_0为小于θ的最大的角，并且$\vec{d}_0=(\cos\theta_0,\sin\theta_0)$是一条边法线，但是，对所有的$\phi\in(\theta_0,\theta]$，$\vec{d}(\phi)=(\cos\phi,\sin\phi)$不是一条边法线。类似地，设$\theta_1$为小于$\theta$的最大的角，并且$\vec{d}_1=(\cos\theta_1,\sin\theta_1)$是一条边法线，但是，对所有的$\phi\in[\theta,\theta_1)$，$\vec{d}(\phi)$不是一条边法线。对所有的方向$\vec{d}(\phi)$（$\phi\in(\theta_0,\theta_1)$），$V_0$是映射为$\lambda_{\max}^{(0)}$的惟一顶点，$V_1$是映射为$\lambda_{\min}^{(1)}$的惟一顶点。连续函数$f(\phi)=(\cos\phi,\sin\phi)\cdot(V_1-V_0)=A\cos(\phi+\psi)$分离这两个区间，其中$A$是常数振幅，$\psi$是常数相角。同时，由于$\vec{d}$是分离方向，因此$f(\theta)>0$。

如果$f(\theta_0)>0$，那么边法线$\vec{d}_0$也是分离方向。如果$f(\theta_1)>0$，那么边法线$\vec{d}_1$也是分离方向。假设$f(\theta_0)\leqslant 0$且$f(\theta_1)\leqslant 0$。由于$f(\theta)>0$，因此f在区间$[\theta_0,\theta_1]$上必定存在两个零值，其中一个值对应的变量值小于θ，另一个则大于θ。f的两个零值被弧度π所分开。这将迫使$\theta_1-\theta_0\geqslant\pi$，此时相连的边的法线$\vec{d}_0$和$\vec{d}_1$之间的角度的弧度至少为$\pi$。只有当该角度刚好为$\pi$时，共享顶点$V_0$的两条边平行于$\vec{d}$，共享顶点$V_1$的两条边平行于$\vec{d}$，这些顶点上的角的对边严格为正。因此，$f(\theta_0)\leqslant 0$和$f(\theta_1)\leqslant 0$不可能同时成立。简单地说，如果$f(\theta)>0$，那么，或者$f(\theta_0)>0$，此时$\vec{d}_0$是一条分离边法线；或者$f(\theta_1)>0$，此时$\vec{d}_1$是一条分离边法线。

图 7.15 说明了$\vec{d}_0$和$\vec{d}_1$的意义。图 7.15（a）显示了两条最近的边法线来自同一个三角形的情形，图 7.15（b）显示了两条最近的边法线来自不同三角形的情形。

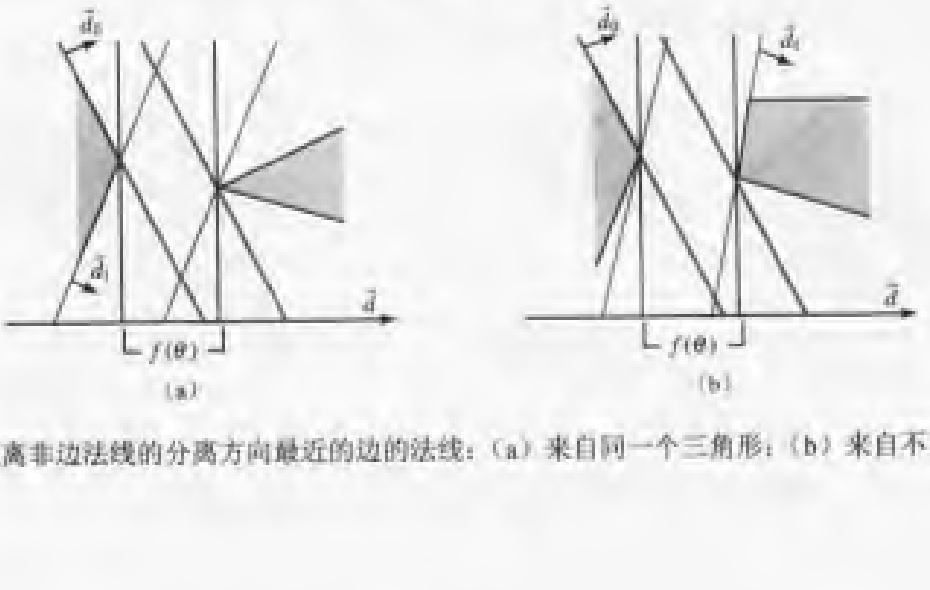


图 7.15　距离非边法线的分离方向最近的边的法线：（a）来自同一个三角形；（b）来自不同的三角形

1. 直接实现

对$\vec{d}$的分离检测的直接实现只需计算投影的极值并比较它们。即计算$\lambda_{\min}^{(j)}(\vec{d}) = \min_{0\le i<N_0}\{\vec{d}\cdot V_i^{(j)}\}$和$\lambda_{\max}^{(j)}(\vec{d}) = \max_{0\le i<N_1}\{\vec{d}\cdot V_i^{(j)}\}$，并检测方程7.10中的不等式。伪码列出如下：

```
bool TestIntersection(ConvexPolygon C0, ConvexPolygon C1)
{
    // test edge normals of C0 for separation
    for (i0 = 0, i1 = C0.N-1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1));  // C0.E(i1) = C0.V(i0) - C0.V(i1)
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if (max1 < min0 || max0 < min1)
            return false;
    }

    // test edge normals of C1 for separation
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1));  // C1.E(i1) = C1.V(i0) - C1.V(i1));
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if (max1 < min0 || max0 < min1)
            return false;
    }

    return true;
}

void ComputeInterval(ConvexPolygon C, Point D, float& min, float& max)
{
    min = max = Dot(D, C.V(0));
    for (i = 1; i < C.N; i++) {
        value = Dot(D, C.V(i));
        if (value < min)
            min = value;
        else if (value > max)
            max = value;
    }
}
```

注意，该实现总是处理包含原点的潜在的分离线。当多边形离原点相对较远时，为了处理浮点误差，上述实现的一种变体可能需要选择潜在的包含多边形顶点的分离线，因而能保持浮点中间值相对较小。

2. 另一种实现

避免将多边形的所有顶点进行投影的另一种实现只需利用第一个多边形的区间的最大值和第二个多边形的区间的最小值来检测分离。如果$\vec{d}$是第一个多边形的边$V_{i+1} - V_i$的指向外面的法线，那么第一个多边形在分离线$V_i + t\vec{d}$上的投影为$[-\mu, 0]$，其中$\mu > 0$。如果

第二个多边形在该直线上的投影为$[\rho_0, \rho_1]$，那么简化的分离检测为$\rho_0 > 0$。图 7.16 说明了用这种方法来对两个多边形进行的分离检测。由于我们仅需将ρ_0与 0 进行比较，因此μ值是无关紧要的，不需要为了计算μ而投影第一个多边形的顶点。而且，第二个多边形的顶点也同时被投影，直到投影值为负为止，此时$\vec{d}$不再是分离线；或者，直到投影值为正为止，此时$\vec{d}$是分离方向。

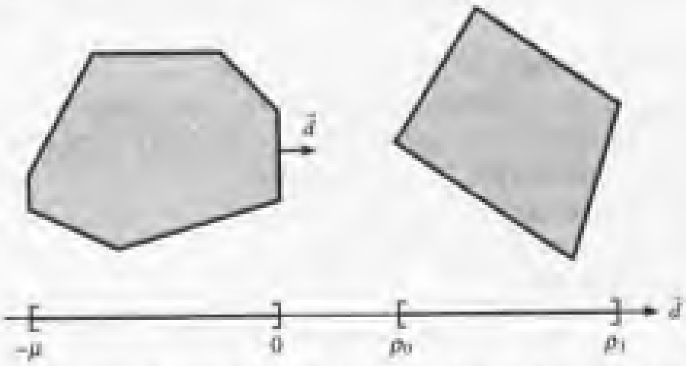


图 7.16　被第一个多边形的一条边的法线方向分离的两个多边形

```
bool TestIntersection(ConvexPolygon C0, ConvexPolygon C1)
{
    // Test edges of C0 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C0 is [m,0] where m <= 0.  Only try to determine
    // if C1 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1));  // C0.E(i1) = C0.V(i0) - C0.V(i1));
        if (WhichSide(C1.V, C0.V(i0), D) > 0) {
            // C1 is entirely on 'positive' side of line C0.V(i0) + t * D
            return false;
        }
    }

    // Test edges of C1 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C1 is [m,0] where m <= 0.  Only try to determine
    // if C0 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1));  // C1.E(i1) = C1.V(i0) - C1.V(i1));
        if (WhichSide(C0.V, C1.V(i0), D) > 0) {
            // C0 is entirely on 'positive' side of line C1.V(i0) + t * D
            return false;
        }
    }

    return true;
}

int WhichSide(PointSet S, Point P, Point D)
{
    // S vertices are projected to the form P + t * D.  Return value is +1 if all t > 0,
    // -1 if all t < 0, 0 otherwise (in which case the line splits the polygon).
```

```
        positive = 0; negative = 0; zero = 0;
        for (i = 0; i < C.N; i++) {
            t = Dot(D, S.V(i) - P);
            if (t > 0) positive++; else if (t < 0) negative++; else zero++;
            if (positive && negative || zero) return 0;
        }
        return positive ? 1 : -1;
    }
```

3. 一种渐近更优的实现

虽然上一种实现比直接实现几乎快两倍，但是它们的耗时数量级都是$O(NM)$，其中N和M为凸多边形的顶点的数目。一种渐近更优的实现使用一种对分法来查找多边形投影的极值点（O'Rourke 1998）。在具有指定方向向量的边的点积的符号发生变化时，对分法将有效地收窄。对于有N个顶点的多边形，对分法的耗时数量级为$O(\log N)$，因此，整个算法的耗时数量级为$O(\max\{N \log M, M \log N\})$。给定多边形的两个顶点$i_0$和$i_1$，这两个顶点的中间索引可以用如下的伪码来计算：

```
int GetMiddleIndex(int i0, int i1, int N)
{
    if (i0 < i1)
        return (i0 + i1) / 2;
    else
        return (i0 + i1 + N) / 2 % N);
    }
```

两个整数的除法返回小于实数比值的最大整数，其中的百分号表示求模运算。注意，如果$i_0 = i_1 = 0$，则该函数返回有效的索引。当$i_0 < i_1$时，其结果是显而易见的：返回的索引是输入的索引的平均值，当然与函数的名称相符。例如，如果多边形有N=5 个顶点，输入$i_0 = 0$和$i_1 = 2$，得到的结果为 1。另一个条件处理顶点重叠的情形。如果$i_0 = 2$和$i_1 = 0$，那么隐含的有序顶点集为$\{2, 3, 4, 0\}$。中间的顶点选为 3，因为$3 = (2 + 0 + 5)/2 \pmod 5$。寻找投影的极值点的对分算法为：

```
int GetExtremeIndex(ConvexPolygon C, Point D)
{
    i0 = 0; i1 = 0;
    while (true) {
        mid = GetMiddleIndex(i0,i1);
        next = (mid + 1) % C.N;
            E = C.V(next) - C.V(mid);
        if (Dot(D, E) > 0)  {
            if (mid != i0) i0 = mid; else return i1;
        } else  {
            prev = (mid + C.N - 1) % C.N;
                E = C.V(mid) - C.V(prev);
            if (Dot(D, E) < 0) i1 = mid; else return mid;
        }
    }
}
```

利用对分方法的相交检测的伪码为：

```
bool TestIntersection(ConvexPolygon C0, ConvexPolygon C1)
{
    // Test edges of C0 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C0 is [m, 0] where m <= 0.  Only try to determine
    // if C1 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++)  {
        D = Perp(C0.E(i1)); // C0.E(i1) = C0.V(i0) - C0.V(i1));
        min = GetExtremeIndex(C1, -D);
        diff = C1.V(min) - C0.V(i0);
        if (Dot(D, diff) > 0) {
            // C1 is entirely on 'positive' side of line C0.V(i0) + t * D
            return false;
        }
    }

    // Test edges of C1 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C1 is [m, 0] where m <= 0.  Only try to determine
    // if C0 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1));  // C1.E(i1) = C1.V(i0) - C1.V(i1));
        min = GetExtremeIndex(C0, -D);
        diff = C0.V(min) - C1.V(i0);
        if (Dot(D,diff) > 0) {
            // C0 is entirely on 'positive' side of line C1.V(i0) + t * D
            return false;
        }
    }

    return true;
}
```

7.7.3 运动凸多边形的分离

轴分离方法可以扩展到以常速运动的凸多边形。Ron Levine 在 GD 算法邮件列表上贴出了该算法（Levine 2000）。如果 C_0 和 C_1 为运动速度为$\vec{w}_0$和$\vec{w}_1$的凸多边形，那么可以通过投影来确定在某些时间$T \geqslant 0$ 时两个多边形是否相交。如果它们相交，那么可以计算它们首次接触的时间。考虑一个固定的多边形 C_0 和一个速度为$\vec{w}$的运动多边形C_1就足够了，因为我们可以通过计算$\vec{w} = \vec{w}_1 - \vec{w}_0$来设定 C_0 是不运动的。

如果初始时C_0和C_1是相交的，那么它们第一次接触的时间为$T=0$。否则，开始时，多边形是分开的。C_1在方向为$\vec{d}$（不垂直于$\vec{w}$）的直线上的投影为其自身的运动。投影沿着直线的运动速度为$\omega = (\vec{w} \cdot \vec{d})/\|\vec{d}\|^2$。如果$C_1$的投影区间远离$C_0$的投影区间，那么两个多边形将不相交。当$C_1$的投影区间趋近$C_0$的投影区间时，两个多边形才有可能相交。

在直观上，如何预测相交在直观上与选择潜在的分离方向很相似。如果两个凸多边形首先相交于时间$T_{\text{first}} > 0$，那么在此时它们的投影沿任何直线都不是分离的。在第一次接触之前的那一刻，多边形是分离的。因此，在时间$T_{\text{first}} - \varepsilon$（对很小的$\varepsilon > 0$），至少必定存在一个分离两个多边形的分离方向。类似地，如果两个多边形最后相交于时间$T_{\text{last}} > 0$，那么

在此时它们的投影沿任何直线都不是分离的，但是在最后一次接触之后的一刻，多边形是分离的。因此，在时间$T_{last}+\varepsilon$（对小的$\varepsilon>0$），必定至少存在一个分离两个多边形的分离方向。当处理每一个潜在的分离轴时，都可计算出T_{first}和T_{last}。当所有方向都处理完后，如果$T_{first}\leqslant T_{last}$，那么两个多边形在第一次接触的时间$T_{first}$确实相交。也可能出现$T_{first}>T_{last}$，在这种情形中，两个多边形不能相交。

下面给出了两个运动的多边形的相交检测的伪码。在实践中，我们感兴趣的时间区间是$[0,T_{max}]$。如果我们希望知道在将来某个时间上存在的相交，那么可设$T_{max}=\infty$。否则，T_{max}是有限的。伪码中的函数设计为说明在$[0,T_{max}]$上不存在相交，尽管在某一时间$T>T_{max}$时，可能存在相交。

```
bool TestIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
    float tmax, float& tfirst, float& tlast)
{
    W = W1 - W0;  // process as if C0 is stationary, C1 is moving
    tfirst = 0;  tlast = INFINITY;

    // test edges of C0 for separation
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1)); // C0.E(i1) = C0.V(i0) - C0.V(i1));
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, min0, max0, min1, max1, tfirst, tlast))
            return false;
    }

    // test edges of C1 for separation
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1)); // C1.E(i1) = C1.V(i0) - C1.V(i1));
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, min0, max0, min1, max1, tfirst, tlast))
            return false;
    }
    return true;
}

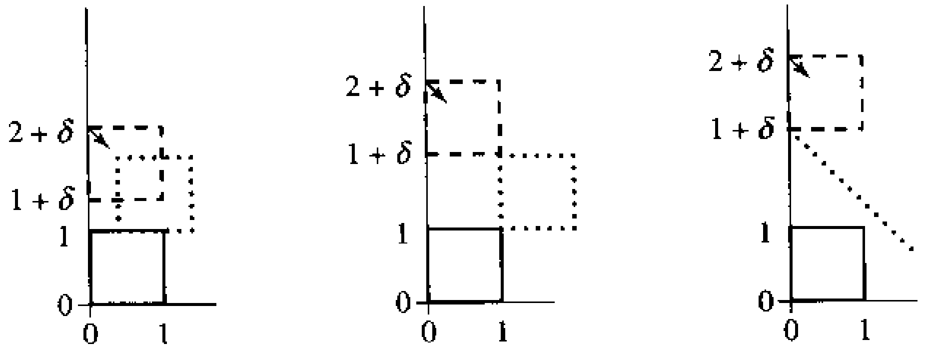
bool NoIntersect(float tmax, float speed, float min0, float max0,
    float min1, float max1, float& tfirst, float& tlast)
{
    if (max1 < min0) {
        // interval(C1) initially on 'left' of interval(C0)
        if (speed <= 0) return true;  // intervals moving apart
        t = (min0 - max1) / speed;  if (t > tfirst) tfirst = t;
        if (tfirst > tmax) return true;
        t = (max0 - min1) /speed;  if (t < tlast) tlast = t;
        if (tfirst > tlast) return true;
```

```
        } else if (max0 < min1) {
            // interval(C1) initially on 'right' of interval(C0)
            if (speed >= 0) return true;  // intervals moving apart
            t = (max0 - min1)/speed;  if ( t > tfirst ) tfirst = t;
            if (tfirst > tmax) return true;
            t = (min0 - max1)/speed;  if ( t < tlast ) tlast = t;
            if (tfirst > tlast) return true;
        }  else {
            // interval(C0) and interval(C1) overlap
            if (speed > 0) {
                t = (max0 - min1) / speed;  if (t < tlast) tlast = t;
                if (tfirst > tlast) return true;
            } else if (speed < 0) {
                t = (min0 - max1) / speed;  if (t < tlast) tlast = t;
                if (tfirst > tlast) return true;
            }
        }
        return false;
    }
```

下面的例子说明了这一思想。第一个矩形是单位立方体 $0 \leqslant x \leqslant 1$ 和 $0 \leqslant y \leqslant 1$，并且是固定的。第二个矩形初始时为 $0 \leqslant x \leqslant 1$ 和 $1+\delta \leqslant y \leqslant 2+\delta$（$\delta > 0$）。设其速度为（1，−1）。第二个矩形与第一个矩形是否相交取决于δ的值。可能的分离轴只有（1，0）和（0，1）。图 7.17 显示了三个 δ 值的初始构形，一个是边一边相交的，一个是顶点一顶点相交的，另一个是不相交的。黑色的矩形是固定的。虚线矩形是运动的。黑色的向量表明运动的方向。点状的矩形说明运动的矩形与固定的矩形第一次相交的位置。在图 7.17（c）中，点线说明运动的矩形将不与固定矩形相交。对于$\vec{d}=(1,0)$，伪码的计算结果为min0 = 0， max0 = 1，min1 = 0，max1 = 1，以及 speed = 1。投影的区间初始时重叠。由于速度为正，T = (max0 - min1) /speed = 1 < TLast = INFINITY，而且TLast更新为 1。对于$\vec{d}=(0,1)$，伪码计算得到的结果为min0 = 0， max0 = 1， min1 = 1 + delta， max1= 2 + delta，以及speed = -1。运动的投影区间初始位于固定的投影区间的右边。由于速度为负，T = (max0 - min1)/speed = delta > TFirst = 0并且TFirst更新为delta。下一段代码设置T = (min0 - max1)/speed = 2 + delta。不更新TLast的值，因为对于$\delta > 0$，不可能出现$2+\delta < 1$。在退出可能的分离方向的循环时，$T_{\text{First}}=\delta$且$T_{\text{last}}=1$。仅当$T_{\text{first}} \leqslant T_{\text{last}}$或$\delta \leqslant 1$时，两个物体相交。该条件与图 7.17 一致。图 7.17（a）中有$\delta < 1$，而图 7.17（b）中有$\delta = 1$；在两种情形中都出现相交。图 7.17（c）中有$\delta > 1$，此时不相交。

7.7.4 固定凸多边形的交集

寻找两个固定的凸多边形的相交问题是多边形布尔运算的一个特例。13.5 节提供了关于计算布尔运算的一般讨论。特别提供了关于凸多边形相交的线性时间计算的讨论。即，如果两个多边形分别具有 N 个和 M 个顶点，相交算法的时间花费为$O(N+M)$级。一种效率较低但可能更易于理解的算法是对照另一个多边形裁剪每一个多边形的边。这种算法的时间花费为$O(NM)$级。当然，对于很大的数 N 和 M 可以进行渐近分析，因此，对三角形和矩形来说后面一种算法可能是一种好的选择。



(a) $0<\delta<1$　　(b) $\delta=1$　　(c) $\delta>1$

图7.17 （a）边一边相交预测 （b）顶点一顶点相交预测 （c）不相交预测

7.7.5 运动凸多边形的接触点集

给定两个运动的凸对象 C_0 和C_1，它们开始时不相交，运动速度分别为 $\vec{w}_0$ 和$\vec{w}_1$，我们已在前面介绍了如果它们即将相交，以及如何计算它们首先相交的时间 T。假设它们相交，集合$C_0+T\vec{w}_0=\{X+T\vec{w}_0:X\in C_0\}$和$C_1+T\vec{w}_1=\{X+T\vec{w}_1:X\in C_1\}$将仅仅互相接触而不贯通。图7.14显示了这类不同的构形。

可以修改函数TestIntersection，以保持跟踪记录哪一个顶点或边将投影到投影区间的端点。在第一次接触时，可以用该信息来确定两个对象是如何相对运动的。如果接触是顶点一边或者顶点一顶点接触，那么接触集只是一个点，即一个顶点。如果接触是边一边接触，那么接触集是至少包含一个顶点的一条线段。投影区间的端点由一个顶点或者一条边产生。双字符的标志标记在每一个多边形上，用来说明投影类型。单字符的标志如下，V表示一个顶点的投影，E表示一条边的投影。4个双字符的标志分别为VV，VE，EV和EE。第一个字符对应于区间的最小值，第二个字符对应于区间的最大值。还必须存储投影区间和投影到区间的极值的分量的顶点或边的索引。一种方便的数据结构为

```
Configuration
{
    float min, max;
    int index[2];
    char type[2];
};
```

其中投影区间为[min, max]。例如，如果投影类型为EV，index[0]就是投影到最小值的边的索引，而index[1]就是投影到最大值的边的索引。

其中声明了两种构形对象，即关于C_0的Cfg0和关于C_1的Cfg1。在TestIntersection中的第一个循环中，由于使用了边的指向外面的法线，C_0在包含顶点V_{i_0}且与$\vec{e}_{i_1}=V_{i_0}-V_{i_1}$垂直的直线上的投影产生一个第二个索引为E的投影类型。第一个索引可能为V或者E，这取决于多边形。伪码如下所示：

```
void ProjectNormal(ConvexPolygon C, Point D, int edgeindex, Configuration Cfg)
{
    Cfg.max = Dot(D, C.V(edgeindex));  // = Dot(D, C.V((edgeindex + 1) % C.N))
```

```
            Cfg.index[1] = edgeindex;
        Cfg.type[0] = 'V';
        Cfg.type[1] = 'E';

        Cfg.min = Cfg.max;
        for (i = 1, j = (edgeindex + 2) % C.N; i < C.N; i++, j = (j + 1) % C.N) {
                value = Dot(D, C.V(j));
            if (value < Cfg.min) {
                Cfg.min = value;
                Cfg.index[0] = j;
            } else if (value == Cfg.min) {
                // Found an edge parallel to initial projected edge.  The
                // remaining vertices can only project to values larger than
                // the minimum.  Keep the index of the first visited end point.
                Cfg.type[0] = 'E';
                return;
            } else { // value > Cfg.min
                // You have already found the minimum of projection, so when
                // the dot product becomes larger than the minimum, you are
                // walking back towards the initial edge.  No point in
                // wasting time to do this, just return since you now know
                // the projection.
                return;
            }
        }
    }
```

C_1在C_0的边的法线上的投影可能为任何类型。伪码为

```
void ProjectGeneral(ConvexPolygon C, Point D, Configuration Cfg)
{
    Cfg.min = Cfg.max = Dot(D, C.V(0));
    Cfg.index[0] = Cfg.index[1] = 0;

    for (i = 1; i < C.N; i++) {
        value = Dot(D, C.V(i));
        if (value < Cfg.min)  {
            Cfg.min = value;
            Cfg.index[0] = i;
        } else if (value > Cfg.max) {
            Cfg.max = value;
            Cfg.index[1] = i;
        }
    }

    Cfg.type[0] = Cfg.type[1] = 'V';
    for (i = 0; i < 2; i++) {
        if (Dot(D, C.E(Cfg.index[i] - 1)) == 0) {
            Cfg.index[i] = Cfg.index[i] - 1;
```

```
            Cfg.type[i] = 'E';
        }  else if (Dot(D, C.E(Cfg.index[i] + 1)) == 0) {
            Cfg.type[i] = 'E';
        }
    }
}
```

C的边的索引运算结果与 C.N 进行模运算，以保证最终的索引位于范围内。

函数 NoIntersect 接受两个多边形的投影区间作为输入参数。既然这些区间存储在构形对象中，因此必须修改 NoIntersect 以反映这一点。在两个运动的多边形将相交的情形中，必须存储构形信息，以备将来用于确定接触集。函数 NoIntersect 必须跟踪记录对应于当前第一次接触的构形对象，并以此作为计算结果。最后，计算接触集将要求投影区间的次序的信息。如果区间相交于C_0的区间的最大值和C_1的区间的最小值，那么函数 NoIntersect 将设置一个值为+1 的标志，或者，如果区间相交于C_0的区间的最小值和C_1的区间的最大值，那么函数 NoIntersect 将设置一个值为−1 的标志。修改后的伪码为

```
bool NoIntersect(float tmax, float speed, Configuration Cfg0,
    Configuration Cfg1, Configuration& Curr0, Configuration& Curr1,
    int& side, float& tfirst, float& tlast)
{
    if (Cfg1.max < Cfg0.min) {
        if (speed <= 0) return true;
        t = (Cfg0.min - Cfg1.max) / speed;
        if (t > tfirst)  {
            tfirst = t;  side = -1;  Curr0 = Cfg0;  Curr1 = Cfg1;
        }
        if (tfirst > tmax  return true;
        t = (Cfg0.max - Cfg1.min) / speed;  if (t < tlast) tlast = t;
        if (tfirst > tlast) return true;
    } else if (Cfg0.max < Cfg1.min) {
        if (speed >= 0) return true;
        t = (Cfg0.max - Cfg1.min) / speed;
        if (t > tfirst)  {
            tfirst = t;  side = +1;  Curr0 = Cfg0;  Curr1 = Cfg1;
        }
        if (tfirst > tmax) return true;
        t = (Cfg0.min - Cfg1.max) / speed;  if (t < tlast) tlast = t;
        if (tfirst > tlast) return true;
    } else {
        if (speed > 0) {
            t = (Cfg0.max - Cfg1.min) / speed;  if (t < tlast) tlast = t;
            if (tfirst > tlast) return true;
        } else if (speed < 0) {
            t = (Cfg0.min - Cfg1.max) / speed;  if (t < tlast) tlast = t;
            if (tfirst > tlast) return true;
        }
    }
    return false;
}
```

根据上述的修改，函数TestIntersection 具有如下的等价形式：

```
bool TestIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
                      float tmax, float& tfirst, float& tlast)
{
    W = W1 - W0;  // process as if C0 stationary and C1 moving
    tfirst = 0;  tlast = INFINITY;

    // process edges of C0
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1)); // = C0.V(i0) - C0.V(i1));
        ProjectNormal(C0, D, i1, Cfg0);
        ProjectGeneral(C1, D, Cfg1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                tlast))
            return false;
    }

    // process edges of C1
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1)); // = C1.V(i0) - C1.V(i1));
        ProjectNormal(C1, D, i1, Cfg1);
        ProjectGeneral(C0, D, Cfg0);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                tlast))
            return false;
    }

    return true;
}
```

函数 FindIntersection的伪码与函数TestIntersection的伪码具有相同的实现，但是，如果存在相交，那么前者在两个循环完成后还有额外的代码段。当多边形将于时间T相交时，它们分别以各自的速度运动，并计算接触集。设$U_i^{(j)} = V_i^{(j)} + T\vec{w}^{(j)}$表示多边形运动后的顶点，在边一边接触的情形中，为了讨论的方便，设边分别为$\vec{e}_0^{(0)}$和$\vec{e}_0^{(1)}$。图 7.18 显示了两个三角形的构形：由于多边形是逆时针次序的，因此两条边互相平行，但是方向相反。第一个多边形的边的参数表示为$U_0^{(0)} + s\vec{e}_0^{(0)}$，$s \in [0,1]$。第二个多边形的边具有相同的参数形式，只是$s \in [s_0, s_1]$，其中

$$s_0 = \frac{\vec{e}_0^{(0)} \cdot \left(U_1^{(1)} - U_0^{(0)}\right)}{||\vec{e}_0||^2} \quad 且 \quad s_1 = \frac{\vec{e}_0^{(0)} \cdot \left(U_0^{(1)} - U_0^{(0)}\right)}{||\vec{e}_0||^2}$$

如果$\bar{s} \in I = [0,1] \cap [s_0, s_1] \neq \emptyset$，那么两条边将出现重叠。在接触集中的对应点为$V_0^{(0)} + T\vec{w}^{(0)} + \bar{s}\vec{e}_0^{(0)}$。

在两个多边形初始重叠的情形中，需要花费更多的时间来建立接触集。可以利用涉及多边形的布尔运算的标准方法来建立这一集合。

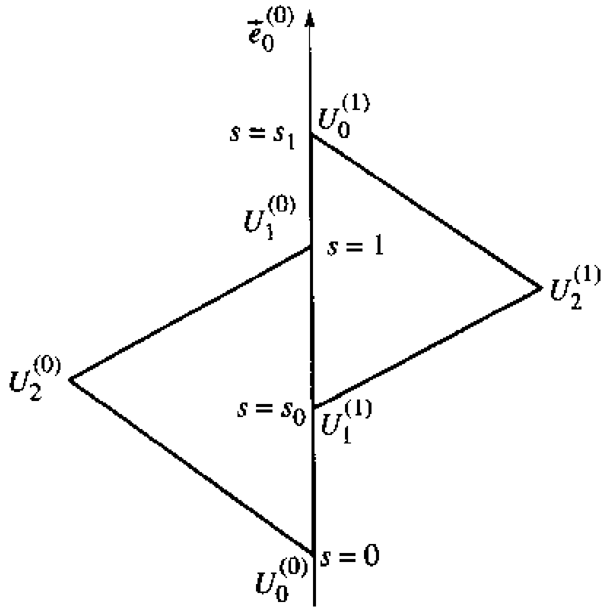


图 7.18 两个运动多边形的边一边接触

其伪码如下所示。交集是一个凸多边形，并通过函数的最后两个参数被返回。如果交集不为空，函数的返回值为true。交集本身必须是凸的。该集合的顶点的数目以quantity存放，逆时针次序的顶点存放在数组I[]中。如果返回值为false，那么函数的最后两个参数无效，不应该使用。

```
bool FindIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
    float tmax, float& tfirst, float& tlast, int& quantity, Point I[])
{
    W = W1 - W0;  // process as if C0 stationary and C1 moving
    tfirst = 0;  tlast = INFINITY;

    // process edges of C0
    for (i0 = 0, i1 = C0.N - 1; i0 < C0.N; i1 = i0, i0++) {
        D = Perp(C0.E(i1)); // C0.E(i1) = C0.V(i0) - C0.V(i1));
        ProjectNormal(C0, D, i1, Cfg0);
        ProjectGeneral(C1, D, Cfg1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                    tlast))
            return false;
    }

    // process edges of C1
    for (i0 = 0, i1 = C1.N - 1; i0 < C1.N; i1 = i0, i0++) {
        D = Perp(C1.E(i1)); // C1.E(i1) = C1.V(i0) - C1.V(i1));
        ProjectNormal(C1, D, i1, Cfg1);
        ProjectGeneral(C0, D, Cfg0);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst,
                    tlast))
```

```
            return false;
        }

        // compute the contact set
        GetIntersection(C0, W0, C1, W1, Curr0, Curr1, side, tfirst, quantity, I);
        return true;
    }
```

计算相交的伪码如下所示。注意其中是如何用投影类型来确定接触是顶点一顶点接触、边一顶点接触还是边一边接触的。

```
void GetIntersection(ConvexPolygon C0, Point W0, ConvexPolygon C1, Point W1,
    Configuration Cfg0, Configuration Cfg1, int side, float tfirst,
    int& quantity, Point I[])
{
    if (side == 1) {  // C0-max meets C1-min
        if (Cfg0.type[1] == 'V') {
            // vertex-vertex or vertex-edge intersection
            quantity = 1;
            I[0] = C0.V(Cfg0.index[1]) + tfirst * W0;
        } else if (Cfg1.type[0] == 'V') {
            // vertex-vertex or edge-vertex intersection
            quantity = 1;
            I[0] = C1.V(Cfg1.index[0]) + tfirst * W1;
        } else {  // Cfg0.type[1] == 'E' && Cfg1.type[0] == 'E'
            // edge-edge intersection
            P = C0.V(Cfg0.index[1]) + tfirst * W0;
            E = C0.E(Cfg0.index[1]);
            U0 = C1.V(Cfg1.index[0]);
            U1 = C1.V((Cfg1.index[0]+ 1) % C1.N);
            s0 = Dot(E, U1 - P) / Dot(E, E);
            s1 = Dot(E,U0 - P) / Dot(E, E);
            FindIntervalIntersection(0, 1, s0, s1, quantity, interval);
            for (i = 0; i < quantity; i++)
                I[i] = P + interval[i] * E;
        }
    } else if (side == -1) { // C1-max meets C0-min
        if (Cfg1.type[1] == 'V') {
            // vertex-vertex or vertex-edge intersection
            quantity = 1;
            I[0] = C1.V(Cfg1.index[1]) + tfirst * W1;
        } else if (Cfg0.type[0] == 'V') {
            // vertex-vertex or edge-vertex intersection
            quantity = 1;
            I[0] = C0.V(Cfg0.index[0]) + tfirst * W0;
        } else { // Cfg1.type[1] == 'E' && Cfg0.type[0] == 'E'
            // edge-edge intersection
            P = C1.V(Cfg1.index[1]) + tfirst * W1;
            E = C1.E(Cfg1.index[1]);
            U0 = C0.V(Cfg0.index[0]);
            U1 = C0.V((Cfg0.index[0] + 1) % C0.N);
```

```
            s0 = Dot(E, U1 - P) / Dot(E, E);
            s1 = Dot(E, U0 - P) / Dot(E, E);
            FindIntervalIntersection(0, 1, s0, s1, quantity, interval);
            for (i = 0; i < quantity; i++)
                I[i] = P + interval[i] * E;
        }
    } else { // polygons were initially intersecting
        ConvexPolygon C0Moved = C0 + tfirst * W0;
        ConvexPolygon C1Moved = C1 + tfirst * W1;
        FindPolygonIntersection(C0Moved, C1Moved, quantity, I);
    }
}
```

当两个多边形初始重叠时，出现最后的情形，因此第一次接触的时间为$T=0$。函数FindPolygonIntersection是计算两个多边形相交的通用过程。

第 8 章　其他二维问题

本章包括一系列关于直线、圆和三角形的问题。它们中大部分都是经常（至少是有时）遇到的问题，另外的一些虽然不那么常见，但是可以用来说明如何利用各种不同的技术来解决新问题。

8.1　三点确定的圆

假定有三个点P_0，P_1和P_2。这三个点惟一地定义一个圆$C:\{C,r\}$，如图 8.1 所示。这个问题等价于寻找由这三个顶点构成的三角形的外接圆，该问题的解答可从 13.10 节中找到。

8.2　与三条直线相切的圆

假定我们有三条直线$\mathcal{L}_0$，$\mathcal{L}_1$和$\mathcal{L}_2$。如果它们中任意两条直线都不平行，那么存在惟一的一个与三条直线都相切的圆$C:\{C,r\}$，如图 8.2 所示。

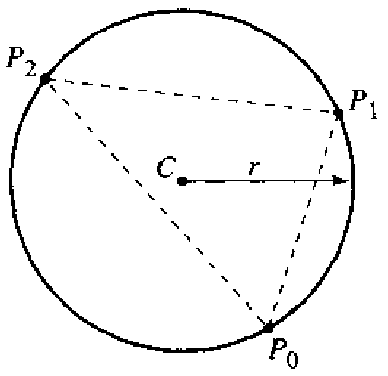


图 8.1　三点确定的圆

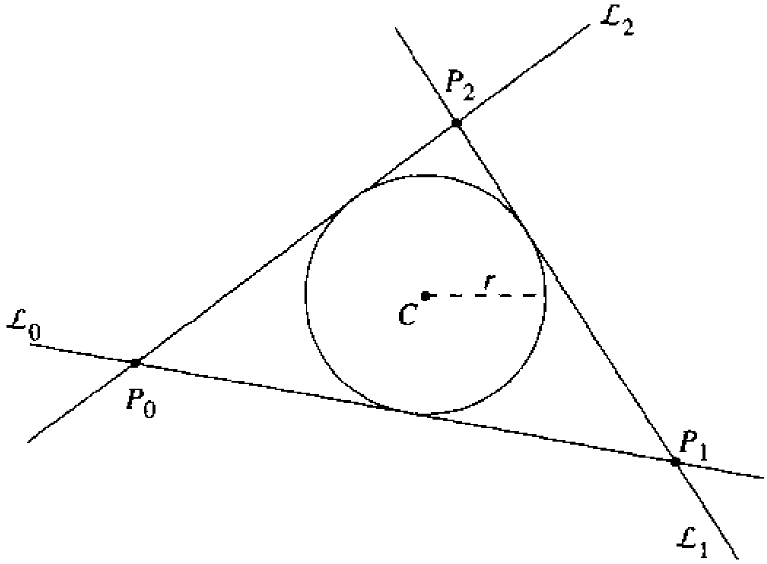


图 8.2　与三条直线相切的圆

三条相交的直线构成一个三角形，因此这一问题等价于寻找一个三角形的内切圆。如果我们首先计算这些交点，那么，我们的问题等价于寻找交点所构成的三角形的内切圆。对该问题的解答可以从 13.10 节中找到。

8.3 与圆相切于给定点的直线

图 8.3 显示了一条通过一个圆上的给定点并且与圆相切的直线。用参数形式来表示该直线是很简单的：

$$\mathcal{L}:\{P,(P-C)^{\perp}\}$$

或者用坐标项来表示：

$$x=P_x-t(P_y-C_y)$$

$$y=P_y+t(P_x-C_x)$$

隐含形式也同样简单：

$$\mathcal{L}:\{P-C,-((P-C)\cdot P)\}$$

伪码为：

```
void LineTangentToCircleAtGivenPoint(Line2D line, Point2D c, Point2D p)
{
    Vector2D v = p - c;
    line.direction.x = -v.y;
    line.direction.y = v.x;
    line.origin = p;
}
```

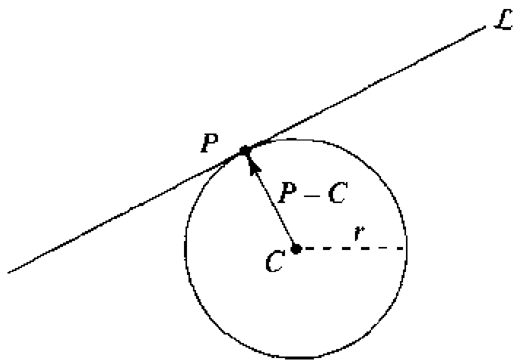


图 8.3 与圆相切于给定点的直线

8.4 通过给定点并与圆相切的直线

假定我们有一个，由其圆心C和半径r所定义的圆C，以及一个点P，我们希望得到通过点P且与圆相切的直线$\mathcal{L}_0$和$\mathcal{L}_1$，如图 8.4 所示。注意，对于任意的点P和圆C，可能存在一条切线、两条切线或没有切线的情形，如图 8.5 所示。

解决这个问题的关键，是注意到$\mathcal{L}_0$ ($\mathcal{L}_1$)的方向向量垂直于C的圆心与圆上的点Q_0 (Q_1)之间的向量$\vec{v}_0$ ($\vec{v}_1$)。考虑到（任一个）$\vec{v}$与$\vec{u}=P-C$之间的角度θ，利用点积的定义，我们得到

$$\cos\theta = \frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\|\vec{v}\|}$$

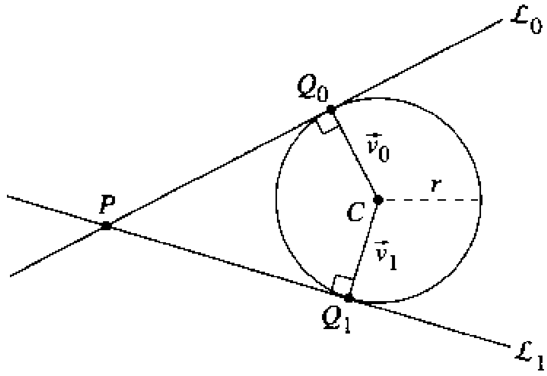


图 8.4 通过一个点并与圆相切的直线

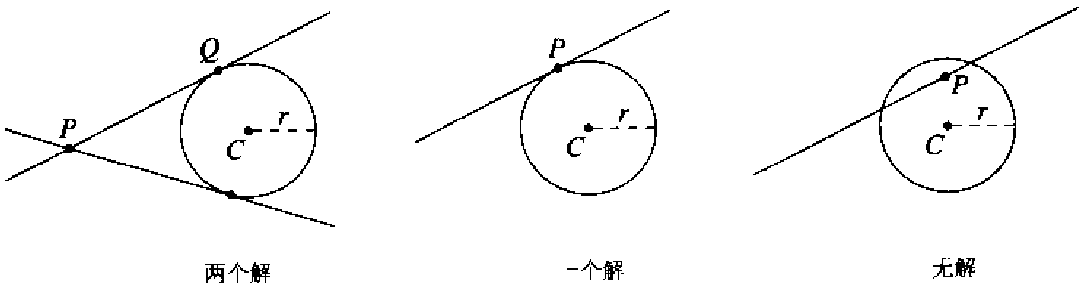


图 8.5 一般有两条切线，但也可能只有一条切线，或者没有切线

然而，通过三角几何，我们也知道

$$\cos\theta = \frac{r}{\|\vec{u}\|}$$

使它们相等，可得

$$\frac{\vec{v}\cdot\vec{u}}{\|\vec{u}\|\|\vec{v}\|} = \frac{r}{\|\vec{u}\|}$$

如果我们注意到$\|\vec{v}\| = r$，我们可将上式简化为

$$\frac{\vec{u}\cdot\vec{v}}{\|\vec{u}\|\|\vec{v}\|} = \frac{r}{\|\vec{u}\|}$$

$$\frac{\vec{u}\cdot\vec{v}}{r\|\vec{u}\|} = \frac{r}{\|\vec{u}\|}$$

$$\vec{u}\cdot\vec{v} = r^2$$

现在我们有如下的两个方程

$$\vec{u}\cdot\vec{v} = r^2$$

$$\|\vec{v}\| = r$$

其中$\vec{u}$和r是已知的，$\vec{v}$包含两个未知数。在分量形式中，我们有

$$u_x v_x + u_y v_y = r^2$$

$$\sqrt{v_x v_x + v_y v_y} = r$$

求解两个未知数v_x和v_y：

$$v_x = \frac{r^2 - \frac{r^2 u_y^2}{u_x^2+u_y^2} \mp \frac{u_y\sqrt{-(r^4 u_x^2)+r^2 u_x^4+r^2 u_x^2 u_y^2}}{u_x^2+u_y^2}}{u_x}$$

$$v_y = \frac{r^2 u_y \pm \sqrt{-\left(r^4 u_x^2\right) + r^2 u_x^4 + r^2 u_x^2 u_y^2}}{u_x^2 + u_y^2}$$

可以通过计算上面方程的+/−和−/+组合来得到两个向量 $\vec{v}_0$ 和$\vec{v}_1$。切线垂直于$\vec{v}_0$和$\vec{v}_1$，因此我们可得如下两条切线：

$$\mathcal{L}_0(t) = P + t\vec{v}_0^{\perp}$$

$$\mathcal{L}_1(t) = P - t\vec{v}_1^{\perp}$$

伪码为

```
int TangentLineToCircleThroughPoint(
    Point2D p,
    float   radius,
    Point2D center,
    Line    solution[2])
{
    int   numSoln;
    float distanceCP;

    distanceCP = dist2D(center,p);

    Vector2D u, v0, v1;

    if (distanceCP < radius) {
        numSoln = 0;
    } else if (distanceCP == radius) {
        numSoln = 1;
        u = p - center;
        solution[0].setDir(-u.y, u.x);
        solution[0].setPoint(p.x, p.y);
    } else if (distanceCP > radius) {
        numSoln = 2;
        u = p - center;
        float ux2 = u.x * u.x;
        float ux4 = ux2 * ux2;
        float uy2 = u.y * u.y;
        float r2  = radius * radius;
        float r4  = r2 * r2;
        float num = r2 * uy2;
        float denom = ux2 + uy2;
```

```
            float rad = sqrt(-(r4 * ux2) + r2 * ux4 + r2 * ux2 * uy2);

            v0.x = (r2 - (num + u.y * rad)/denom)/u.x
            v0.y = (r2 * u.y) + rad)/ denom;
            v1.x = (r2 - (num - u.y * rad)/denom)/u.x
            v1.y = (r2 * u.y) - rad)/ denom;

            solution[0].setDir(-v0.y, v.x);
            solution[0].setPoint(p.x, p.y);

            solution[1].setDir(v1.y, -v1.x)
            solution[1].setPoint(p);

            // Note: may wish to normalize line directions
            // before returning, depending on application
        }
        return numSoln;
    }
```

8.5 与两圆相切的直线

给定两个圆，我们希望找到一条直线同时与这两个圆相切，如图 8.6 所示。这两个圆分别用它们的圆心和半径来表示：$\{C_0, r_0\}$ 和$\{C_1, r_1\}$。正如从图 8.7 中看到的一样，一般存在四条、两条、一条、没有或无数条切线。在我们的解决方法中，我们假定两个圆既不互相包含，也不相交。如果$\|C_1 - C_0\| > r_0 + r_1$，则能满足上述条件。

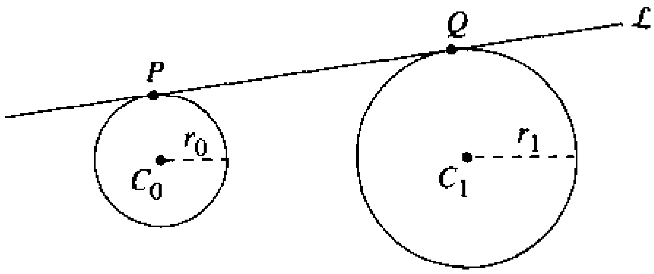


图 8.6　与两个圆相切的直线

我们将采用参数形式来求解直线，$\mathcal{L}: X(t) = P + t\hat{d}$，并假定$r_0 \geq r_1$。与第一个圆相交的直线满足

$$r_0^2 = \|X(t) - C_0\|^2 = t^2 + 2(\hat{d} \cdot (P - C_0))t + \|P - C_0\|^2 \tag{8.1}$$

对于相切于交点的直线，应该同时满足

$$0 = \hat{d} \cdot (X(t) - C_0) = t + \hat{d} \cdot (P - C_0) \tag{8.2}$$

如果我们求解方程（8.2）中的 t，并将其代回方程（8.1），可以得到

$$r_0^2 = \|P - C_0\|^2 - (\hat{d} \cdot (P - C_0))^2 \tag{8.3}$$

对第二个圆应用相同的步骤，可得

$$r_1^2 = \|P - C_1\|^2 - (\hat{d} \cdot (P - C_1))^2 \tag{8.4}$$

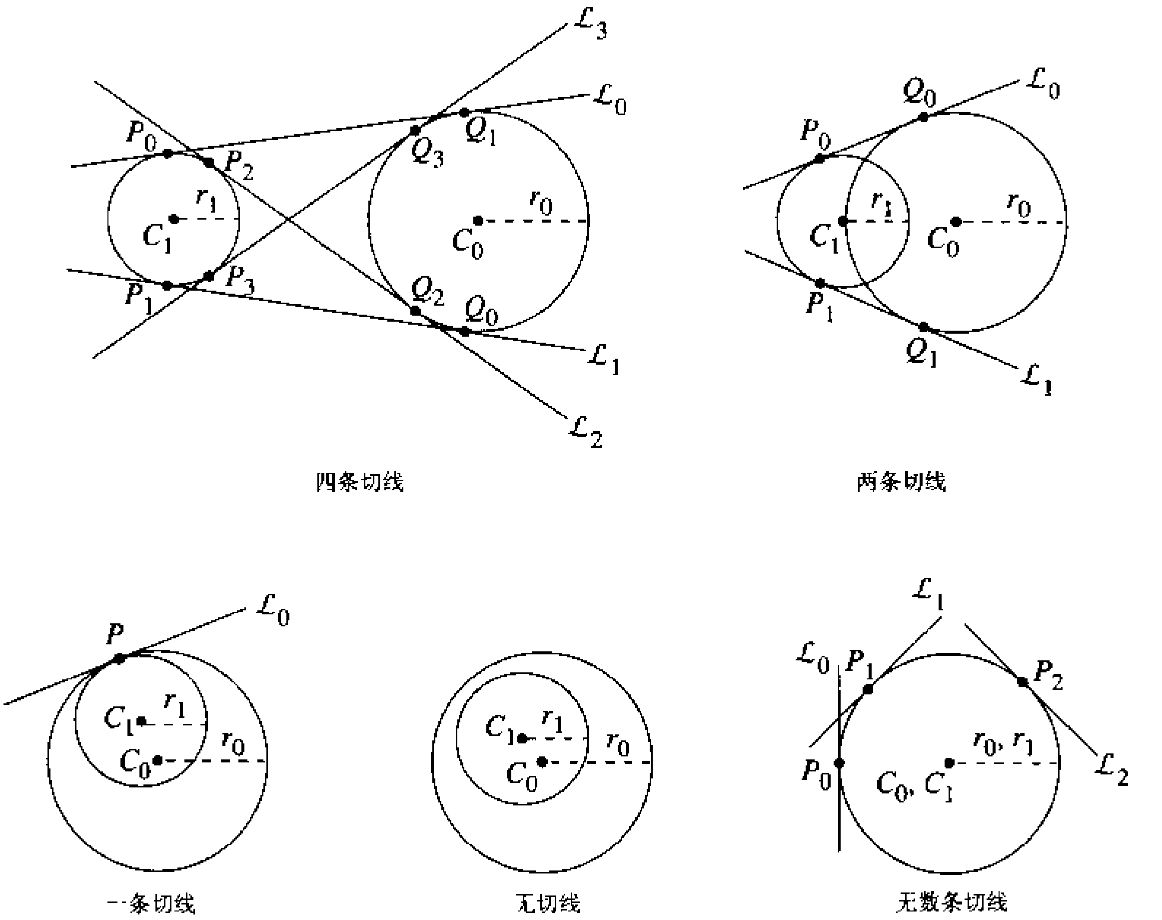


图 8.7 两圆之间的切线数目随圆的相对大小和位置不同而变化

点 P 可以从包含 C_0 和 C_1（两个圆心）的直线上选取。设 $P=(1-s)C_0+sC_1$ 对某些 s 成立，并设 $\vec{w}=C_1-C_0$，那么

$$P-C_0=s\vec{w}$$

$$P-C_1=(s-1)\vec{w}$$

如果将它们代回方程（8.3）和方程（8.4），可得

$$r_0^2=s^2(\|\vec{w}\|^2-(\hat{d}\cdot\vec{w})^2)$$

$$r_1^2=(s-1)^2(\|\vec{w}\|^2-(\hat{d}\cdot\vec{w})^2)$$

因此 $r_0^2/s^2=r_1^2/(s-1)^2$，或者

$$(r_1^2-r_0^2)s^2+2r_0^2s-r_0^2=0$$

如果两个圆有相同的半径（$r_0=r_1$），那么 $s=1/2$，因此，P 是 C_0 和 C_1（两个圆的圆心）之间的线段的中点。而且

$$(\hat{d}\cdot\vec{w})^2=\|\vec{w}\|^2-4r_0^2=a^2>0$$

因此 $\hat{d}\cdot\vec{w}=a$（自然，我们可以用 $-a$ 来作为根，这只是得到一个符号相反的方向向量）。如果我们设 $\hat{d}=(d_0,d_1)$，那么 $\hat{d}\cdot\vec{w}=a$ 是一条直线的方程。约束条件 $\|\hat{d}\|^2=1$ 对应于一个圆。

这两个条件表示一条直线与一个圆的相交。任何一个解都成立。

设$\vec{w}=(w_0, w_1)$。那么$w_0 d_0 + w_1 d_1 = a$且$d_0^2 + d_1^2 = 1$。如果$|w_0| \geq |w_1|$，那么$d_0 = (a - w_1 d_1)/w_0$且

$$(w_0^2 + w_1^2)d_1^2 - 2a w_1 d_1 + a^2 - w_0^2 = 0$$

如果$|w_1| \geq |w_0|$，那么$d_1 = (a - w_0 d_0)/w_1$且

$$(w_0^2 + w_1^2)d_0^2 - 2a w_0 d_0 + a^2 - w_1^2 = 0$$

在任一种情形下，两个根都得到切线的两个方向向量。

如果$r_0 > r_1$，那么关于s的二次曲线有两个实数解。从几何上来说，其中的一个值必须满足$0 < s < 1$且产生两条相交于两个圆之间的切线（图 8.7 中"四条切线"情形中的$\mathcal{L}_2$和$\mathcal{L}_3$）。另一个根不能为$s = 0$（否则P将位于圆心，而这是不可能的）。

对每一个根s，可以得到与$r_1 = r_0$的情形相似的结论。求解的二次曲线为

$$(\hat{d} \cdot \vec{w})^2 = \|\vec{w}\|^2 - r_0^2/s_2 = a^2 > 0$$

同时

$$(\hat{d} \cdot \vec{w})^2 = \|\vec{w}\|^2 - r_1^2/(s-1)^2 = a^2 > 0$$

当$s^2 \geq (s-1)^2$时，应该应用第一个方程。否则，应用第二个方程。可相对d_0或d_1建立相同的二次曲线（a取不同的值）并求解。

伪码为

```
void GetDirections(
    Vector2D  w,
    double    a,
    Vector2D& dir0,
    Vector2D& dir1)
{
    double aSqr = a * a;
    double wxSqr = w.x * w.x;
    double wySqr = w.y * w.y;
    double c2 = wxSqr + wySqr, invc2 = 1.0 / c2;
    double c0, c1, discr, invwx, invwy;

    if (fabs(w.x) >= fabs(w.y)) {
        c0 = aSqr - wxSqr;
        c1 = -2.0 * a * w.y;
        discr = sqrt(fabs(c1 * c1 - 4.0 * c0 * c2));
        invwx = 1.0 / w.x;
        dir0.y = -0.5 * (c1 + discr) * invc2;
        dir0.x = (a - w.y * dir0.y) * invwx;
        dir1.y = -0.5 * (c1 - discr) * invc2;
        dir1.x = (a - w.y * dir1.y) * invwx;
    } else {
        c0 = aSqr -w ySqr;
        c1 = -2.0 * a * w.x;
        discr = sqrt(fabs(c1 * c1 - 4.0 * c0 * c2));
```

```
            invwy = 1.0 / w.y;
            dir0.x = -0.5 * (c1 + discr) * invc2;
            dir0.y = (a - w.x * dir0.x) * invwy;
            dir1.x = -0.5 * (c1 - discr) * invc2;
            dir1.y = (a - w.x * dir1.x) * invwy;
        }
    }

    int LinesTangentToTwoCircles(
        Circle2D circle0,
        Circle2D circle1,
        Line2D line[4])

        Vector2D w = { circle1.center.x - circle0.center.x,
                       circle1.center.y - circle0.center.y };
        double wLenSqr = w.x * w.x + w.y * w.y;
        double rSum = circle0.radius + circle1.radius;
        if (wLenSqr <= rSum * rSum) {
            return 0;  // circles are either intersecting or nested
        }

        double epsilon = 1e-06;
        double rDiff = circle1.radius - circle0.radius;
        if (fabs(rDiff) >= epsilon) {
            // solve (R1^2-R0^2)*s^2 + 2*R0^2*s - R0^2 = 0.
            double R0sqr = circle0.radius * circle0.radius;
            double R1sqr = circle1.radius * circle1.radius;
            double c0 = -R0sqr;
            double c1 = 2.0 * R0sqr;
            double c2 = circle1.radius * circle1.radius - R0sqr, invc2 = 1.0 / c2;
            double discr = sqrt(fabs(c1 * c1 - 4.0 * c0 * c2));
            double s, oms, a;

            // first root
            s = -0.5 * (c1 + discr) * invc2;
            line[0].p.x = circle0.center.x + s * w.x;
            line[0].p.y = circle0.center.y + s * w.y;
            line[1].p.x = line[0].p.x;
            line[1].p.y = line[0].p.y;
            if (s >= 0.5) {
                a = sqrt(fabs(wLenSqr - R0sqr / (s * s)));
            } else {
                oms = 1.0-s;
                a = sqrt(fabs(wLenSqr - R1sqr / (oms * oms)));
            }
            GetDirections(w, a, line[0].direction, line[1].direction);

            // second root
            s = -0.5 * (c1 - discr) * invc2;
            line[2].p.x = circle0.center.x + s * w.x;
            line[2].p.y = circle0.center.y + s * w.y;
```

```
            line[3].p.x = line[2].p.x;
            line[3].p.y = line[2].p.y;
            if (s >= 0.5) {
                a = sqrt(fabs(wLenSqr - R0sqr / (s * s)));
            } else {
                oms = 1.0 - s;
                a = sqrt(fabs(wLenSqr - R1sqr / (oms * oms)));
            }
            GetDirections(w, a, line[2].direction, line[3].direction);
        } else {
            // circles effectively have same radius

            // midpoint of circle centers
            Point2 mid =
            {
                0.5 * (circle0.center.x + circle1.center.x),
                0.5 * (circle0.center.y + circle1.center.y)
            };

            // tangent lines passing through midpoint
            double a = sqrt(fabs(wLenSqr - 4.0 * circle0.radius * circle0.radius));
            GetDirections(w, a, line[0].direction, line[1].direction);
            line[0].p.x = mid.x;
            line[0].p.y = mid.y;
            line[1].p.x = mid.x;
            line[1].p.y = mid.y;

            // unitize w
            double invwlen = 1.0 / sqrt(wLenSqr);
            w.x *= invwlen;
            w.y *= invwlen;

            // tangent lines parallel to unitized w
            //   1.  D = w
            //   2.  a. P = mid + R0 * perp(w), perp(a, b) = (b, -a)
            //       b. P = mid - R0 * perp(w)
            line[2].p.x = mid.x + circle0.radius * w.y;
            line[2].p.y = mid.y - circle0.radius * w.x;
            line[2].direction.x = w.x;
            line[2].direction.y = w.y;
            line[3].p.x = mid.x - circle0.radius * w.y;
            line[3].p.y = mid.y + circle0.radius * w.x;
            line[3].direction.x = w.x;
            line[3].direction.y = w.y;
        }

        return 1;
    }
```

8.6 两点和给定半径决定的圆

给定两个不重合的点 P 和 Q，我们想找到一个经过这两个点的圆。当然，实际上有无数个圆经过这两个点，因此我们必须指定一个半径 r，如图 8.8 所示。与我们讨论过的其他问题一样，存在不止一个解。实际上，存在两个这样的圆，如图 8.9 所示。

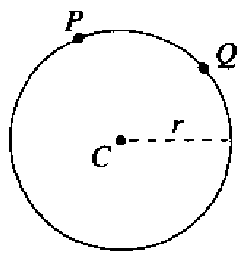


图 8.8　两点和给定半径决定的圆

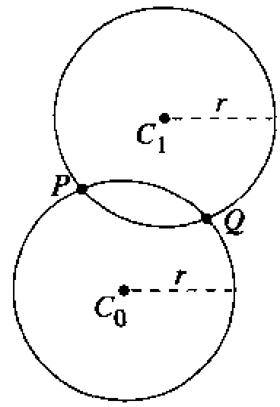


图 8.9　两点和给定半径决定的两个圆

进一步研究这个问题可以发现，满足要求的圆的圆心位于半径为 r，圆心分别位于 P 和 Q 的两个圆的交点，如图 8.10 所示。也就是说，我们只需创建两个半径为 r，圆心分别位于 P 和 Q 的圆，计算出它们的交点，即为两个半径为 r，并通过 P 和 Q 的圆的圆心。两个圆的交点的求解方法可在 7.5.2 节中找到。

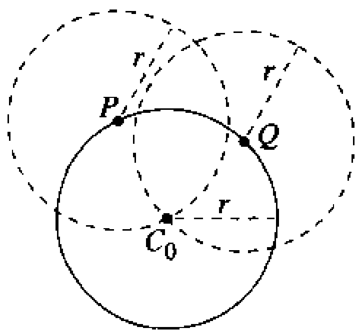


图 8.10　求解由两点和给定半径决定的圆

伪码为

```
CircleThrough2PointsGivenR(Point2D p1, Point2D p2, float radius,
                           Point2D centers[2])
{
    // See Section 7.5.2
    FindIntersectionOf2DCircles(p1, p2, radius, radius, centers);
}
```

8.7 通过一点并与一条直线相切且具有给定半径的圆

假定我们有一条直线$\mathcal{L}$和一个点P。本节的问题是，找到一个具有给定的半径r，与直线相切且经过指定点的圆，如图 8.11 所示。当然，实际上存在两个（可能的）圆满足要求，如图 8.12 所示。

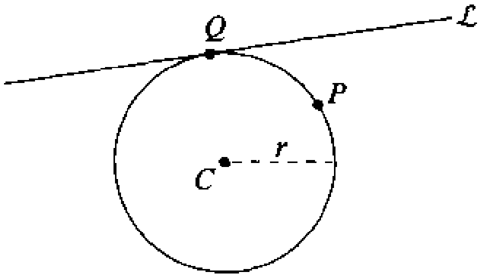


图 8.11 经过一点并与一条直线相切且具有给定半径的圆

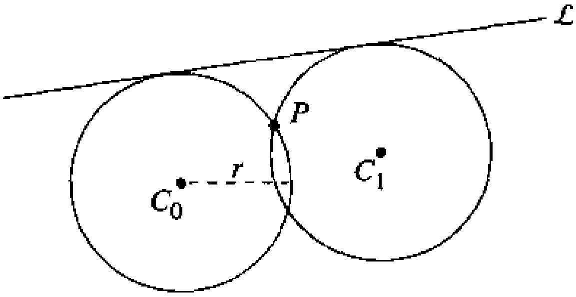


图 8.12 一般有两个不同的圆经过给定点

其他的可能构形是点P位于直线上，或者点P与直线$\mathcal{L}$的距离大于$2r$。对于第一种情形，有两种解，它们分别位于直线的两侧。在第二种情形中，无解，如图 8.13 所示。

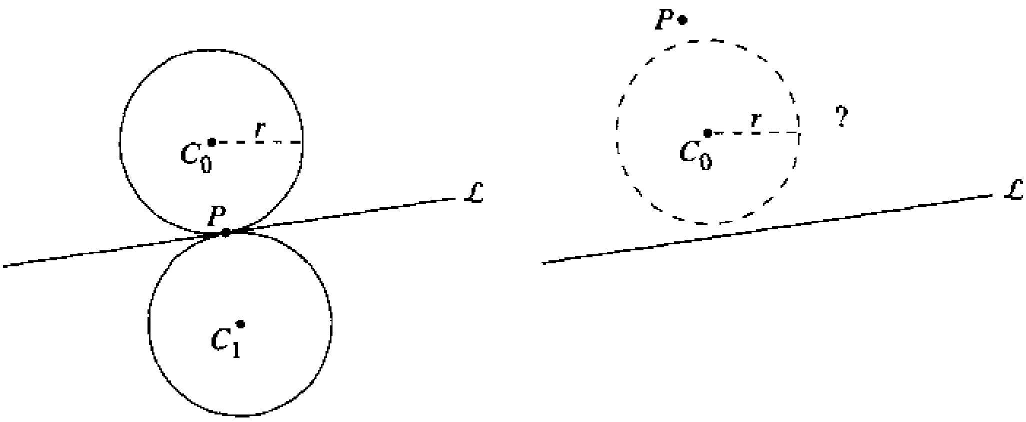


图 8.13 如果P在直线上，两个圆将以直线为轴成镜像；如果P与直线的距离大于圆的直径，则无解

进一步研究该问题可以发现，圆C的圆心C必须位于与$\mathcal{L}$距离为r的地方。而且，除了P位于$\mathcal{L}$上的情形，C与P必须位于直线的同一侧。

如果直线用隐含形式给出$\mathcal{L}: ax + by + c = 0$，那么$P$与$\mathcal{L}$之间的（可见）距离为

$$r = \frac{aC_x + bC_y + c}{\sqrt{a^2 + b^2}}$$

我们也知道圆必须经过点P，因此该点必须满足圆的方程：

$$(C_x - P_x)^2 + (C_y - P_y)^2 = r^2$$

一般地说，我们就得到了两个具有两个未知数的方程。我们当然可以很简单地求解这两个方程中的$\{C_x, C_y\}$，得到的解为

$$\begin{aligned} C_x = {} & \frac{1}{a\left(a^2+b^2\right)^{\frac{3}{2}}} a^4 r + ab^2\sqrt{a^2+b^2}P_x \\ & - b\sqrt{a^2+b^2}\sqrt{-\left(a^2\left(c+aP_x+bP_y\right)\left(c-2\sqrt{a^2+b^2}r+aP_x+bP_y\right)\right)} \\ & - a^2\left(\sqrt{a^2+b^2}c + b\left(-(br)+\sqrt{a^2+b^2}P_y\right)\right) \end{aligned} \tag{8.5}$$

$$C_y = \frac{-\left(b\left(c-\sqrt{a^2+b^2}r+aP_x\right)\right)+a^2P_y+\sqrt{-\left(a^2\left(c+aP_x+bP_y\right)\left(c-2\sqrt{a^2+b^2}r+aP_x+bP_y\right)\right)}}{a^2+b^2} \tag{8.6}$$

有许多的方法可以求解上式（参见 Chasen 1978，Bowyer 和 Woodwark 1983）。根据 Bowyer 和 Woodwark（1983），我们平移整个系统使P位于原点，这当然并不会改变$\mathcal{L}$的系数a和b，而仅仅会改变常数c：

$$c' = c + aP_x + bP_y$$

然后我们检查c'的符号。如果为负，我们对方程两边乘以-1（我们可以使c为任意的负值，然后在随后的方程中进行调整）。如果$\mathcal{L}$是规范的，那么方程（8.5）和方程（8.6）可简化为

$$C_x = -a(c' - r) \pm b\sqrt{-c'^2 + 2c'r}$$

$$C_y = -b(c' - r) \mp a\sqrt{-c'^2 + 2c'r}$$

伪码为

```
int CircleThroughPointTangentToLineGivenRadius(
    Point2D point,
    Line2D  line,
    float   radius,
    Point2D center[2])
{
    // Returns number of solutions

    //  Translate line so point is at origin
    float cPrime = line.c + line.a * point.x + line.b * point.y;

    //  Check if point lies on, or nearly on, the line
    if (Abs(cPrime) < epsilon) {
        Vector2D tmp = { line.a, line.b };
        center[0] = point + tmp * r;
```

```
        center[1] = point - tmp * r;
        return 2;
    }
    float a;
    float b;
    float c;
    if (cPrime < 0) {
        // Reverse line
        a = -line.a;
        b = -line.b;
        c = -line.c;
    } else {
        a = line.a;
        b = line.b;
        c = line.c;
    }

    float tmp1 = cPrime - radius;
    float tmp2 = r * r + tmp1 * tmp1;
    if (tmp2 < -epsilon) {
        // No solutions - point further away from
        // line than radius.
        return 0;
    }

    if (tmp2 < epsilon) {
        // One solution only
        center[0].x = point.x - a * tmp1;
        center[0].y = point.y - b * tmp1;
        return 1;
    }

    // Otherwise, two solutions
    tmp2 = Sqrt(tmp2);
    Point2D tmpPt = { point.x - a * tmp1, point.y - b * tmp1 };
    center[0] = { tmpPt + b * tmp2, tmpPt - a * tmp2 };
    center[1] = { tmpPt - b * tmp2, tmpPt + a * tmp2 };
    return 2;
}
```

8.8 与两条直线相切且具有给定半径的圆

假定我们有两条不平行的直线$\mathcal{L}_0$和$\mathcal{L}_1$。可以画出一个具有给定半径r且与这两条直线相切的圆$\mathcal{C}$，如图8.14所示。自然，实际上存在4个这样的圆，如图8.15所示。

给定两条直线$\mathcal{L}_0$和$\mathcal{L}_1$，以及一个半径r，我们的问题是找到圆的圆心C_0，C_1，C_2和C_3。进一步的研究可从如下的事实开始，即每一个圆心C_i与直线$\mathcal{L}_0$和$\mathcal{L}_1$的距离都是r。如果C_i与直线$\mathcal{L}_0$之间的距离为r，那么它必定位于一条与$\mathcal{L}_0$平行且相距r的直线上；如果C_i与直线$\mathcal{L}_1$之间的距离为r，那么它必定位于一条与$\mathcal{L}_1$平行且相距r的直线上。

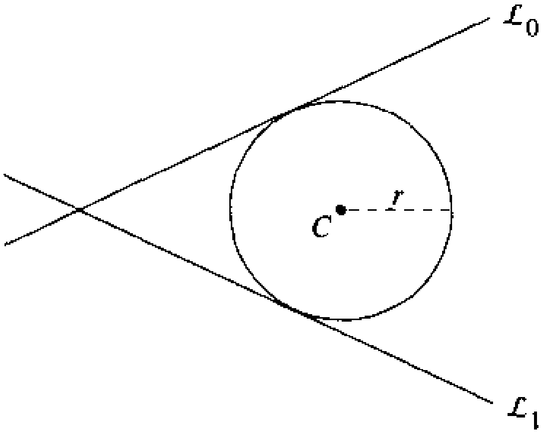


图 8.14　与两条直线相切并具有给定半径的圆

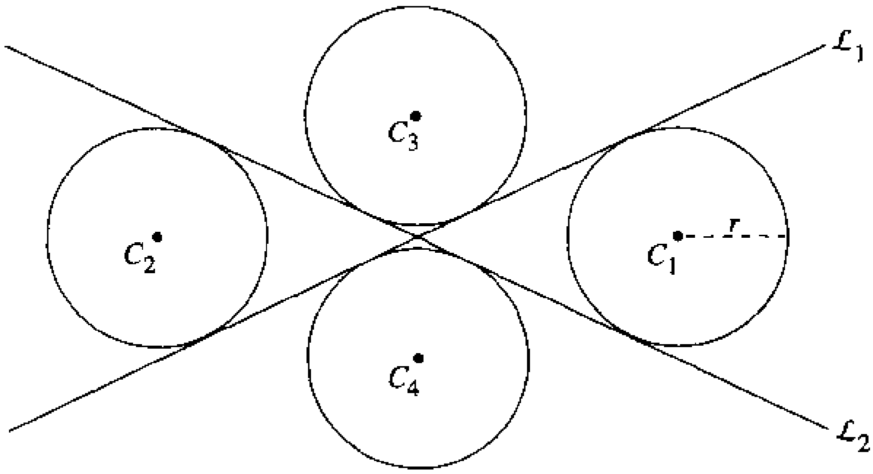


图 8.15　一般存在 4 个具有给定半径并与两条直线相切的圆

因此，圆心 C_i 必定位于两条分别与 $\mathcal{L}_0$ 和 $\mathcal{L}_1$ 平行且相距 r 的直线的交点，图 8.16 显示了 4 个圆中的一个圆。所有的 4 个相切圆都可通过两条平行于 $\mathcal{L}_0$ 和两条平行于 $\mathcal{L}_1$ 的直线的交点对来建立。

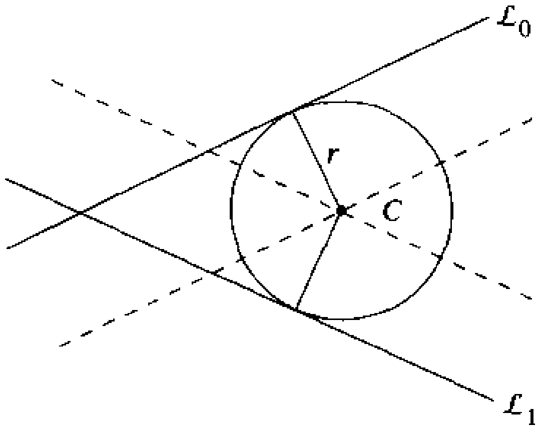


图 8.16　与两条直线相切的圆的求解方法

如果两条直线隐式地定义为

$$\mathcal{L}_0: a_0x + b_0y + c_0 = 0$$

$$\mathcal{L}_1: a_1x + b_1y + c_1 = 0$$

那么，与它们平行且相距r的直线为

$$\mathcal{L}_0: a_0x + b_0y + c_0 \pm \sqrt{a_0^2 + b_0^2} = 0$$

$$\mathcal{L}_1: a_1x + b_1y + c_1 \pm \sqrt{a_1^2 + b_1^2} = 0$$

如果我们求解其中的x 和 y，可得圆心

$$x = -\frac{b_1\left(c_0 \pm \sqrt{a_0^2 + b_0^2}\,r\right) - b_0\left(c_1 \pm \sqrt{a_1^2 + b_1^2}\,r\right)}{-a_1b_0 + a_0b_1}$$

$$y = -\frac{-a_1\left(c_0 \pm \sqrt{a_0^2 + b_0^2}\,r\right) + a_0\left(c_1 \pm \sqrt{a_1^2 + b_1^2}\,r\right)}{-a_1b_0 + a_0b_1}$$

伪码为

```
void CircleTangentToLinesGivenR(Line2D l0, Line2D l1, float radius, Point2D center[4])
{
    float discrm0 = sqrt(l0.a * l0.a + l0.b * l0.b) * r;
    float discrm1 = sqrt(l1.a * l1.a + l1.b * l1.b) * r;
    float invDenom = 1.0 / (-l1.a * l0.b + l0.a * l1.b);

    center[0].x = -(l1.b * (l0.c + discrm0) - l0.b * (l1.c + discrm1)) * invDenom;
    center[0].y = -(l1.a * (l0.c + discrm0) - l0.a * (l1.c + discrm1)) * invDenom;

    center[1].x = -(l1.b * (l0.c + discrm0) - l0.b * (l1.c + discrm1)) * invDenom;
    center[1].y = -(l1.a * (l0.c - discrm0) - l0.a * (l1.c - discrm1)) * invDenom;

    center[2].x = -(l1.b * (l0.c - discrm0) - l0.b * (l1.c - discrm1)) * invDenom;
    center[2].y = -(l1.a * (l0.c - discrm0) - l0.a * (l1.c - discrm1)) * invDenom;

    center[3].x = -(l1.b * (l0.c - discrm0) - l0.b * (l1.c - discrm1)) * invDenom;
    center[3].y = -(l1.a * (l0.c + discrm0) - l0.a * (l1.c + discrm1)) * invDenom;
}
```

8.9 经过一点并与一个圆相切且具有给定半径的圆

给定一个圆$\mathcal{C}_0:\{C_0, r_0\}$和一个点$P$，本节的问题是找到一个具有给定的半径、经过一个给定的点且与给定的圆相切的圆$\mathcal{C}_1:\{C_1, r_1\}$（如图 8.17 所示）。和与切线相关的典型问题一样，一般存在两个解。如果P与C_0之间的距离大于$r_0 + 2r$或小于$r_0 - 2r$，那么可能无解。这取决于圆的相对大小和点P的位置，其中一个圆可能包含另一个圆，因此，自然，可能有四个解、两个解或无解（如图 8.18 所示）。这个问题是很有趣的，因为（至少）可以用两种完全不同的方法来求解，其中一种方法更具解析性，另一种方法更具实用性。

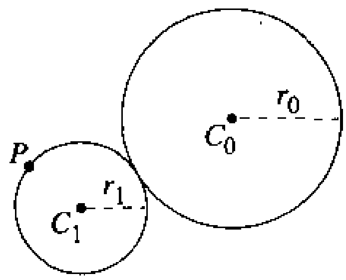


图 8.17　经过一点并与一个圆相切且具有给定半径的圆

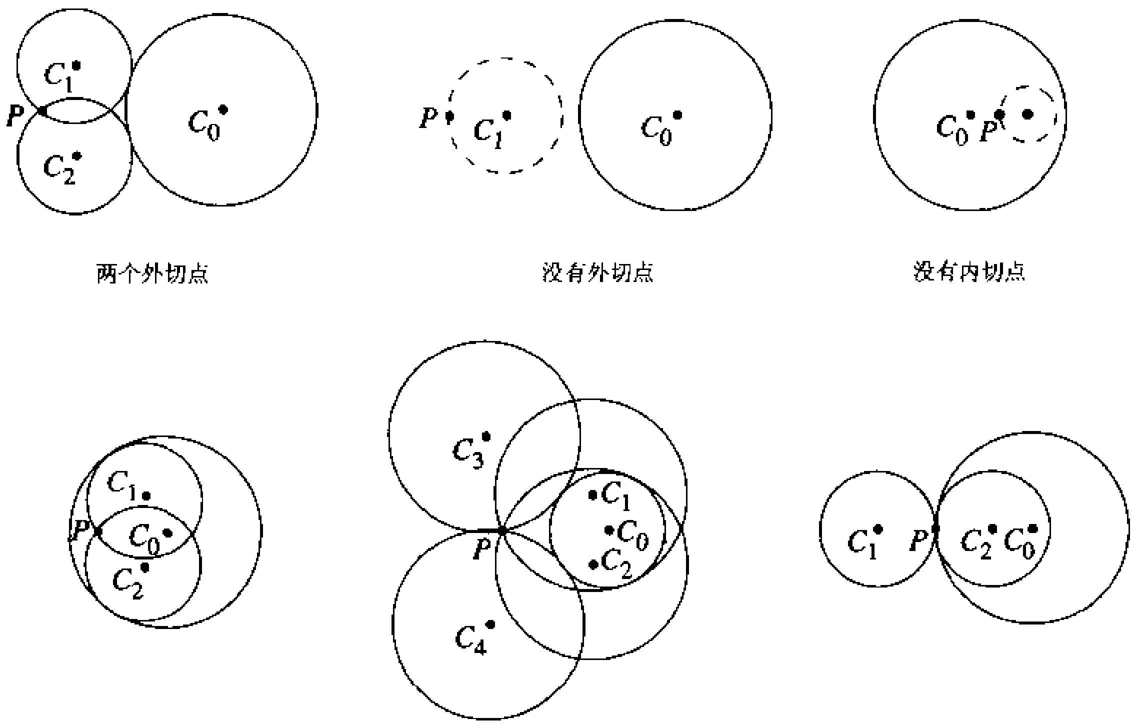


图 8.18　可能存在四个圆、两个圆或无解，随圆的相对位置和半径的不同而变化

更具“解析性”的方法基于如下事实：我们已知三角形（P, C_0, C_1）（如图 8.19 所示）。很显然，如果圆C_0与C_1相切，那么$\|C_1 - C_0\| = r_0 + r_1$。给定点 P 在圆上，因此 $\|P - C_1\| = r_1$。最后，P 和C_0是给定的已知条件。注意，即使 P 位于C_0上，此时三角形退化为一条直线，该方法依然有效。

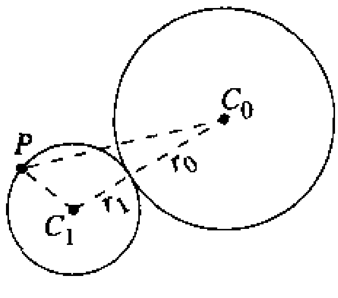


图 8.19　更具解析性的解法

为了减少对平方根函数的调用，我们可以选择考虑距离平方。为了进一步简化方程，我们可以平移系统，使其中一个点位于原点，然后求解，并将解平移回去。我们一般选择将 P 平移到原点。这将产生一个由两个具有两个未知数$\{C_{1,x}, C_{1,y}\}$的方程决定的系统

$$\begin{aligned}(-C_{0,x}+C_{1,x})^2+(-C_{0,y}+C_{1,y})^2&=(r_0+r_1)^2\\ C_{1,x}^2+C_{1,y}^2&={r_1}^2\end{aligned} \tag{8.7}$$

其解（已经平移回去）为

$$\begin{aligned}C_{1,x}&=P_x+\frac{-({r_0}^2{C_{0,x}}^2)-2r_0r_1{C_{0,x}}^2+{C_{0,x}}^4+{C_{0,x}}^2{C_{0,y}}^2\mp C_{0,y}k}{2C_{0,x}({C_{0,x}}^2+{C_{0,y}}^2)}\\ C_{1,y}&=P_y+\frac{-({r_0}^2C_{0,y})-2r_0r_1C_{0,y}+{C_{0,x}}^2C_{0,y}+{C_{0,y}}^3\pm k}{2({C_{0,x}}^2+{C_{0,y}}^2)}\end{aligned} \tag{8.8}$$

其中

$$k=\sqrt{-({C_{0,x}}^2(-{r_0}^2+{C_{0,x}}^2+{C_{0,y}}^2)(-{r_0}^2-4r_0r_1-4{r_1}^2+{C_{0,x}}^2+{C_{0,y}}^2))}$$

伪码为

```
int CircleThroughPTangentToC(Point2D p, Circle2D cIn, float r1, Circle2D cOut[4]
{
    float distanceCPS = Distance2D(p, cIn.center);
    int numSoln;
    if (distanceCP > cIn.radius + 2 * r1) {
        numSoln = 0;
    } else if (distanceCP < cIn.radius - 2 * r1) {
        numSoln = 0;
    } else {
        numSoln = 4;
        float k = sqrt(-(cIn.x^2 * (-cIn.radius^2 + cIn.x^2 + cIn.y^2)) *
            (-cIn.radius^2 - 4 * cIn.radius * r1 - 4 * r1^2 + cIn.x^2 + cIn.y^2));
        float invDenom = 1.0 / (2 * (cIn.x * cIn.x + cIn.y * cIn.y));

        float temp1 = -(cIn.radius^2 * cIn.x^2 ) -2 * cIn.radius * r1 * cIn.x ^2
                    + cIn.x^4 + cIn.x^2 * cIn.y^2;

        float temp2 = -(cIn.radius^2 * cIn.y) -2 * cIn.radius * r1 * cIn.y
                    + cIn.x^2 cIn.y + cIn.y^3;

        cOut[0].x = (p.x + (temp1 - cIn.y * k) * invDenom) / cIn.x;
        cOut[0].y = (p.y + (temp2 + k) * invDenom);

        cOut[1].x = (p.x + (temp1 + cIn.y * k) * invDenom) / cIn.x;
        cOut[1].y = (p.y + (temp2 - k) * invDenom);

        k = -k;

        cOut[2].x = (p.x + (temp1 - cIn.y * k) * invDenom) / cIn.x;
```

```
        cOut[2].y = (p.y + (temp2 + k) * invDenom);

        cOut[3].x = (p.x + (temp1 + cIn.y * k) * invDenom) / cIn.x;
        cOut[3].y = (p.y + (temp2 - k) * invDenom);
    }

    // Note: all solutions not necessarily unique - calling routine
    // should check...
    return numSoln;
}
```

更具"实用性"的方法也是基于更具"解析性"的方法所使用的事实，即$\|C_1 - C_0\| = r_0 + r_1$和$\|P - C_1\| = r_1$。考虑图 8.20。如果我们画一个半径为 r_1，圆心为P的圆，那么，很显然，它将包含C_1。如果我们再画一个半径为$r_0 + r_1$，圆心为 C_0 的圆，那么它也将包含C_1。因此，C_1位于这两个圆的交点（第二个解以另一个交点为圆心）。

求解方法几乎就与我们所说的一样简单。问题演变为简单地寻找两个"辅助圆"的交点。惟一复杂的地方是，你必须考虑P是否真正的位于 C_0 内。如果是，我们画的圆心为C_0的"辅助圆"必须具有半径$r_0 - r_1$（如图 8.21 所示）。有两种情形，即P位于C_0 内或C_0 外。可以相对简单地进行区分：如果比较P和C_0与r_0^2的距离平方，就能避免不必要的平方根运算。

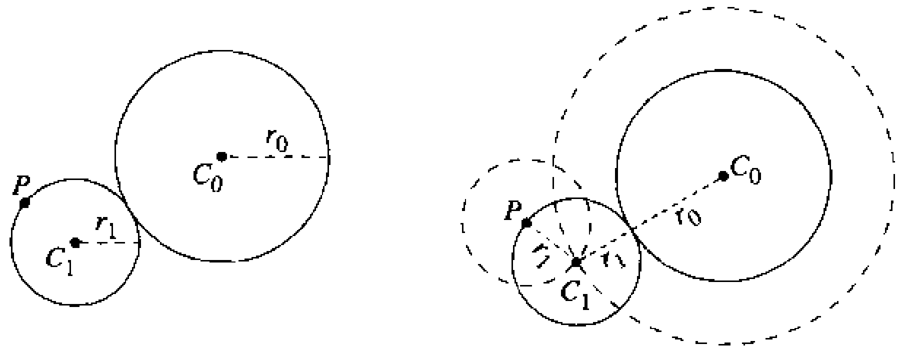


图 8.20　更具实用性的方法

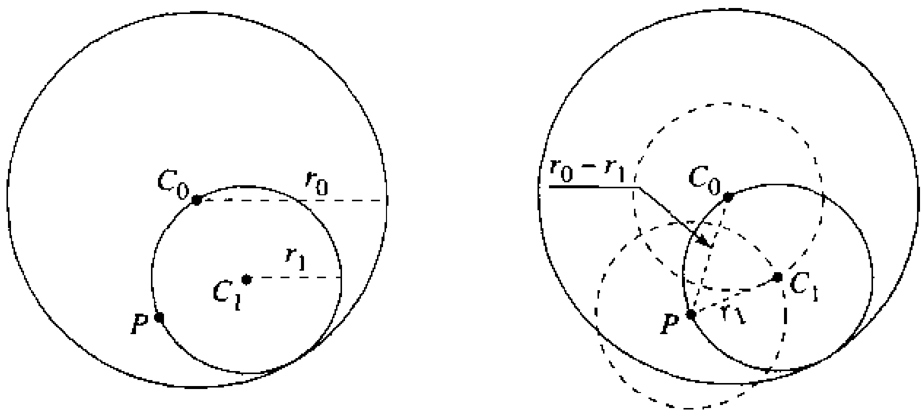


图 8.21　求解问题的实用性方法的特例

8.10 具有给定半径并与一条直线和一个圆相切的圆

假定我们有一个圆$\mathcal{C}_0:\{C_0, r_0\}$和一条直线 $\mathcal{L}: ax + by + c = 0$，我们希望找到同时与该圆和该直线相切且具有给定半径的圆，如图 8.22 所示。当然，可能存在多于一个的解。事实上，可能存在多达 8 个的不同圆，这可从图 8.23 中看出。也可能无解，当 $\mathcal{C}_0$ 与 $\mathcal{L}$ 之间的距离大于$2r_1 + r_0$时，就会出现这种情形，如图 8.24 所示。

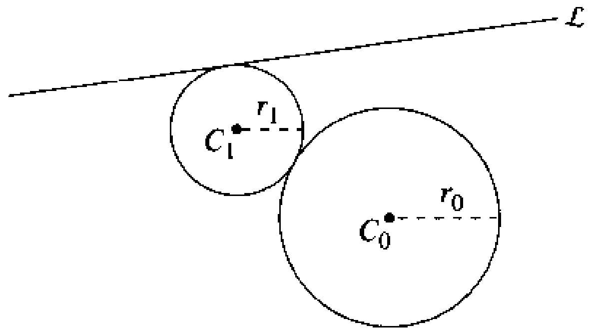


图 8.22 具有给定半径并与一条直线和一个圆相切的圆

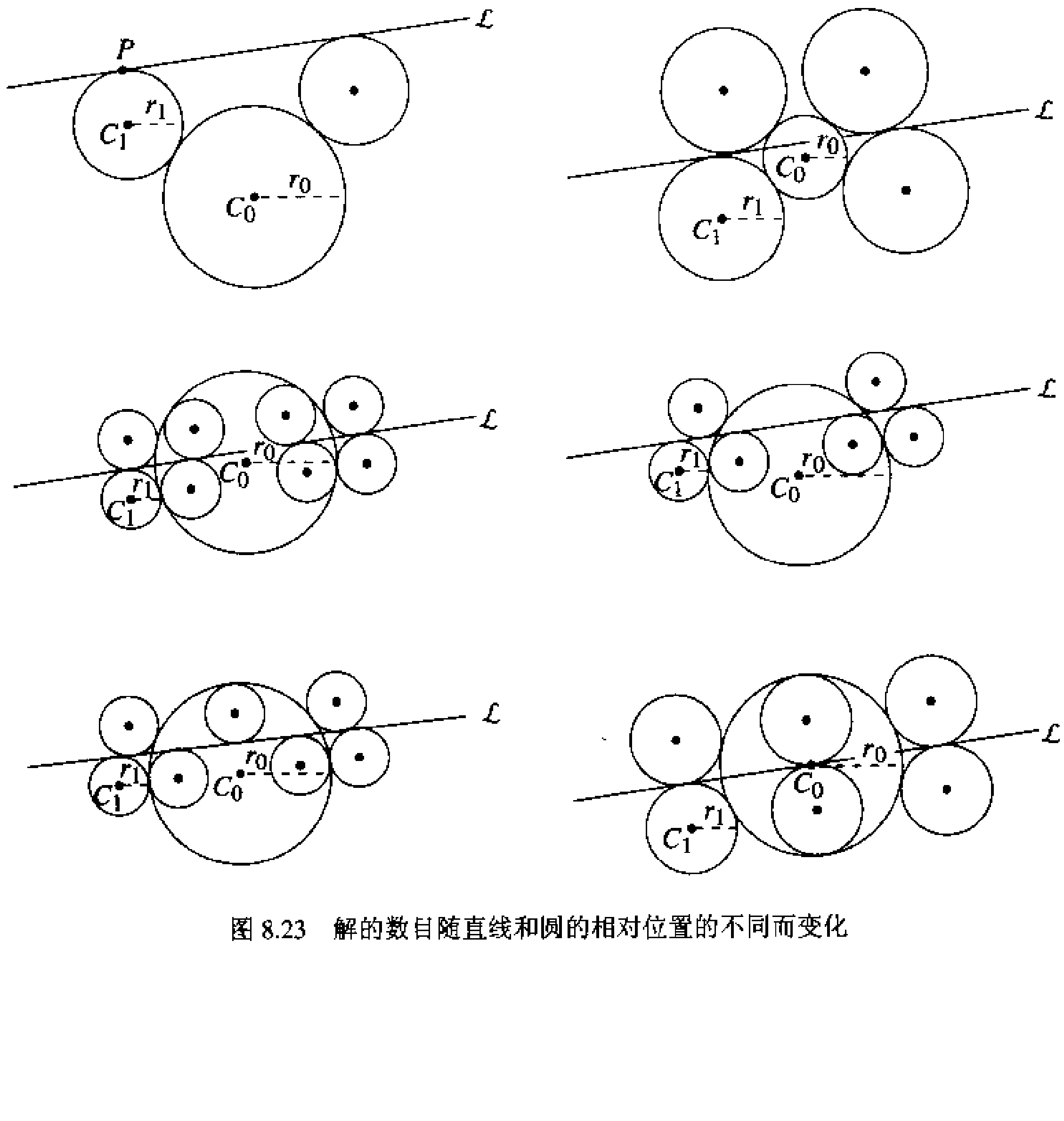


图 8.23 解的数目随直线和圆的相对位置的不同而变化

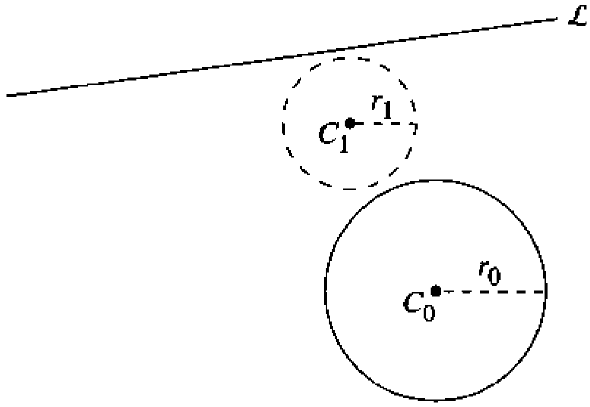


图 8.24　如果给定的半径太小，则无解

进行深入的研究时，应该注意到，如果圆C'与$\mathcal{L}$相切，则圆心C_1与$\mathcal{L}$的距离为r_1，因此，C_1位于一条与$\mathcal{L}$平行且相距r_1的直线$\mathcal{L}'$上，如图 8.25 所示。而且，如果与 C_0 相切，那么圆心C_1与圆心C_0之间的距离为 r_0+r_1。换句话说，C_1必定位于一个圆上，即$C':\{C_0,r_0+r_1\}$。同时与$\mathcal{L}$和C相切的圆的圆心位于$\mathcal{L}'$和C'的交点上（如图 8.26 所示）。

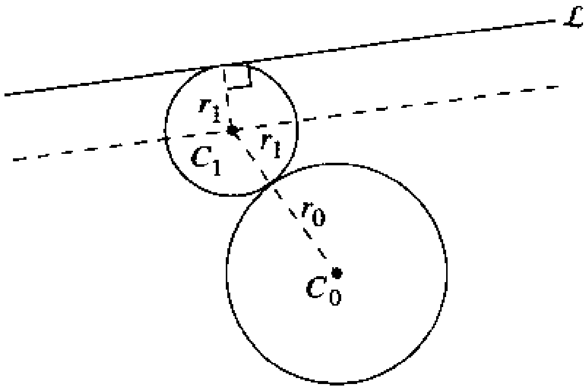


图 8.25　求解具有给定半径的圆的解析

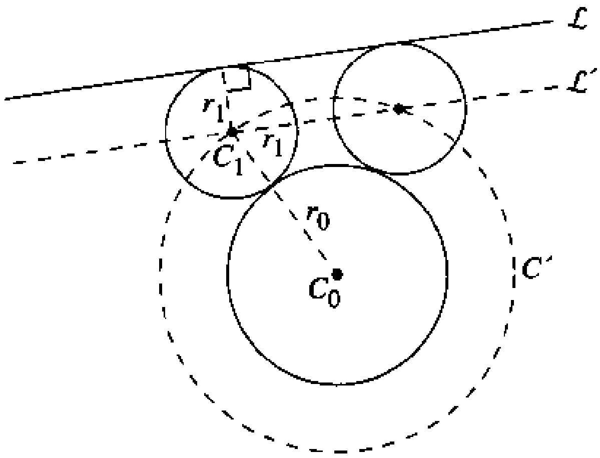


图 8.26　解的示意图

为了获得图 8.23 中的全部 8 个可能的相切圆，我们需要生成外切于C_0 和内切于C_0 的

圆。内切圆的圆心位于L'与圆心为C_0的圆的交点，半径为$r_0 - r_1$。

所有这些似乎都说明进行上述计算是非常复杂的。然而，如果我们采用 Bowyer 和 Woodwark（1983）描述的“诀窍”，那么计算将简单得多。他们的方法是：将所有的对象平移，使P_0位于原点，然后求解目标圆的圆心，并将圆心平移回去。平移是非常简单的，只需将其圆心加到其方程中

$$(x - C_{0,x} + C_{0,x})^2 + (y - C_{0,y} + C_{0,y})^2 + c = r_0^2$$

原方程为

$$x^2 + y^2 = r_0^2$$

如果我们的直线的隐含形式为$ax + by + c = 0$，平移后的直线自然具有相同的系数a和b，新的常数为

$$c' = c + aC_{0,x} + bC_{0,y}$$

如果$c' < 0$，那么，我们在方程的两边同时乘以-1。

C_1的圆心的最终方程为

$$C_{1,x} = C_{0,x} + \frac{a(c' - r_1) \pm b\sqrt{(a^2 + b^2)(r_1 \pm r_0)^2 - (c' - r_1)^2}}{a^2 + b^2}$$

$$C_{1,y} = C_{0,y} + \frac{b(c' - r_1) \mp a\sqrt{(a^2 + b^2)(r_1 \pm r_0)^2 - (c' - r_1)^2}}{a^2 + b^2}$$

伪码为

```
int CirclesTangentToLineAndCircleGivenRadius(
    Line2D   l,
    Circle2D c,
    float    radius,
    Circle2D soln[8])
{
    if (l.distanceToPoint(c.center) > 2 * radius + c.radius){
        return 0;
    } else {
        // Some of these solutions may be duplicates.
        // It is up to the application to deal with this.
        float a, b, c;
        l.getImplicitCoeffs(a,b,c);

        for (i = 0 ; i < 8 ; i++){
            soln.radius = radius;
        }
        float apbSqr = a^2 + b^2;
        float cp = c + a * c.center.x + b * c.center.y;

        float discrm1 = sqrt(apbSqr * (radius + c.radius)^2 - (cp - radius)^2);
        float discrm2 = sqrt(apbSqr * (radius - c.radius)^2 - (cp - radius)^2);
        float cpminusr = cp - radius;
```

```
        soln[0].center.x = c.center.x + (b * cpminusr + b * discrml) / apbSqr;
        soln[0].center.y = c.center.y + (a * cpminusr - a * discrml) / apbSqr;

        soln[1].center.x = c.center.x + (b * cpminusr - b * discrm2) / apbSqr;
        soln[1].center.y = c.center.y + (a * cpminusr + a * discrm2) / apbSqr;

        soln[2].center.x = c.center.x + (b * cpminusr + b * discrm2) / apbSqr;
        soln[2].center.y = c.center.y + (a * cpminusr + a * discrm2) / apbSqr;

        soln[3].center.x = c.center.x + (b * cpminusr - b * discrm2) / apbSqr;
        soln[3].center.y = c.center.y + (a * cpminusr - a * discrm2) / apbSqr;

        soln[4].center.x = c.center.x + (b * cpminusr + b * discrml) / apbSqr;
        soln[4].center.y = c.center.y + (a * cpminusr + a * discrml) / apbSqr;

        soln[5].center.x = c.center.x + (b * cpminusr - b * discrml) / apbSqr;
        soln[5].center.y = c.center.y + (a * cpminusr - a * discrml) / apbSqr;

        soln[6].center.x = c.center.x + (b * cpminusr + b * discrm2) / apbSqr;
        soln[6].center.y = c.center.y + (a * cpminusr - a * discrm2) / apbSqr;

        soln[7].center.x = c.center.x + (b * cpminusr - b * discrml) / apbSqr;
        soln[7].center.y = c.center.y + (a * cpminusr + a * discrml) / apbSqr;
        return 8;
    }
}
```

8.11 具有给定半径并与两圆相切的圆

假定有两个圆$C_0:\{C_0,r_0\}$和$C_1:\{C_1,r_1\}$，我们希望找到一个同时与这两个圆相切且具有给定半径的圆，如图 8.27 所示。当然，可能存在一系列的解，这取决于圆的相对位置，它们的半径，以及为其他圆指定的半径，如图 8.28 所示。

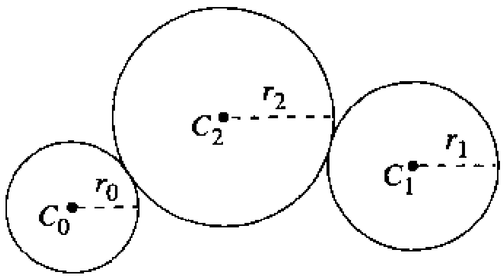


图 8.27　具有给定半径并与两圆相切的圆

我们的第三个圆$C_2:\{C_2,r_2\}$具有已知的半径，我们的任务是计算它的圆心。这个圆必须与C_0和C_1相切，即其圆心与C_0的距离为r_0+r_2，其圆心与C_1的距离为r_1+r_2。进一步分析可知，这一问题等价于寻找圆心位于C_0和C_1，半径分别为r_0+r_2和r_1+r_2的两个圆的交点，如图 8.29 所示。如果我们的初始的圆为

$$C_0:\quad (x-C_{0,x})^2+(y-C_{0,y})^2=r_0^2$$

$$C_1:\quad (x-C_{1,x})^2+(y-C_{1,y})^2=r_1^2$$

那么，两个新的圆的方程为

$$C_0':\quad (x-C_{0,x})^2+(y-C_{0,y})^2=(r_0+r_2)^2$$

$$C_1':\quad (x-C_{1,x})^2+(y-C_{1,y})^2=(r_1+r_2)^2$$

如果计算出 C_0' 和 C_1' 的交点，我们就将得到与它们相切的圆的圆心。7.5.2 节介绍了求解两个圆的交点的方法。

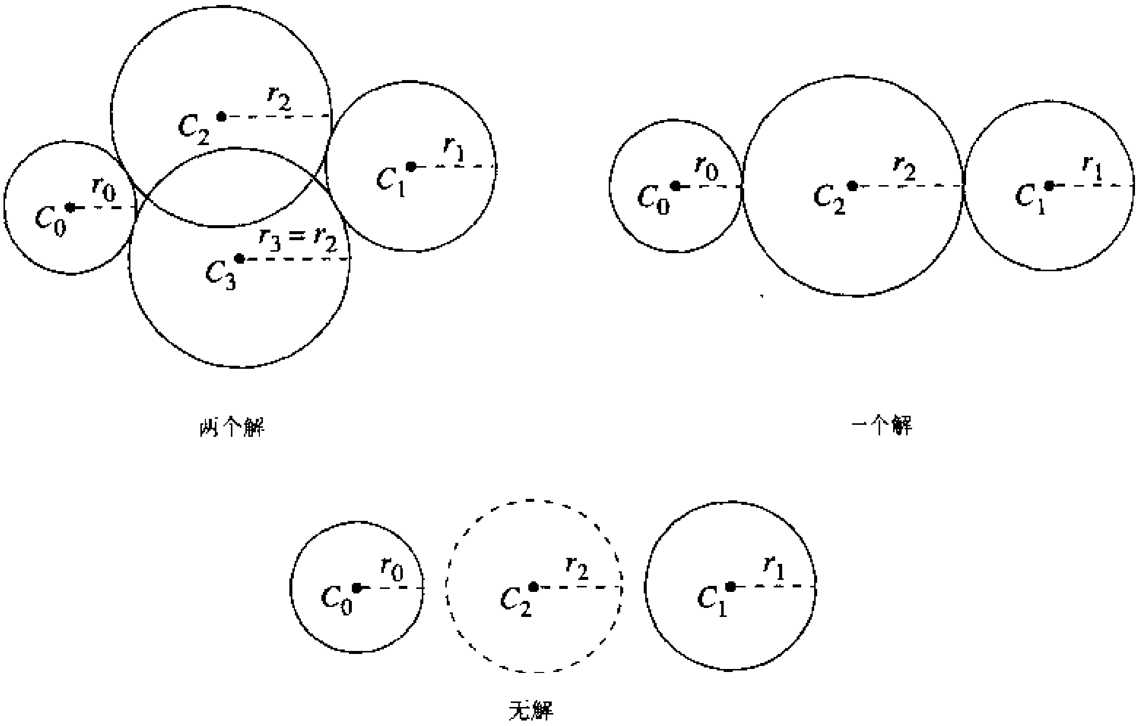


图 8.28　一般存在两个解，但是解的数量随给定圆的相对大小和位置的不同而变化

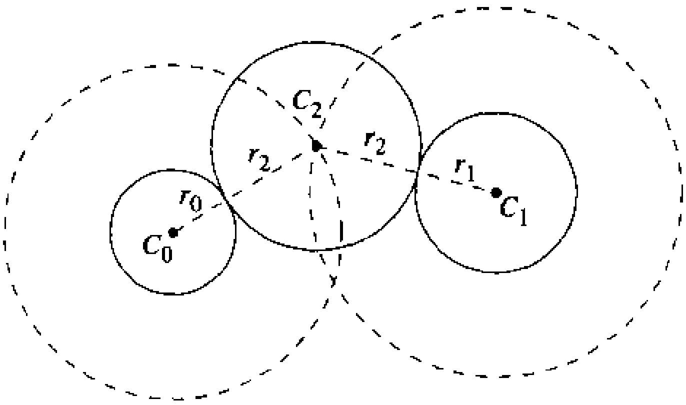


图 8.29　与两圆相切的圆的求解

伪码为

```
int CircleTangentToCirclesGivenRadius(
    Circle2D c1,
    Circle2D c2,
    float radius,
    Circle2D c[2])
{
    Vector2D v = c2.center - c1.center;
    float dprod =  Dot(v, v);
    float dSqr = dprod - (c1.radius + c2.radius)^2;
    if (dSqr > radius^2) {
        // No solution
        return 0;
    } else if (dSqr == radius^2) {
        float distance = sqrt(dprod);
        c.center.x = c1.center.x + (c1.radius + radius) * v.x / distance;
        c.center.y = c1.center.y + (c1.radius + radius) * v.y / distance;
        c.radius = radius;
        return 1;
    } else {
        Circle2D cp1;
        Circle2D cp2;
        cp1.center.x = c1.center.x;
        cp1.center.y = c1.center.y;
        cp1.center.radius = c1.radius + radius;
        cp2.center.x = c2.center.x;
        cp2.center.y = c2.center.y;
        cp2.center.radius = c2.radius + radius;
        // Section 7.5.2
        findIntersectionOf2DCircles(c1, c2, c);
        c[0].radius = radius;
        c[1].radius = radius;
        return 2;
    }
}
```

8.12 与一条给定直线垂直并通过一个给定点的直线

假定我们有一条直线 $\mathcal{L}_0$ 和一个点 Q。我们的问题是找到一条通过点 Q 且与 $\mathcal{L}_0$ 垂直的直线，如图 8.30 所示。如果 $\mathcal{L}_0$ 隐式地表示为 $a_0x+b_0y+c_0=0$，那么 $\mathcal{L}_1$ 的方程为

$$b_0x-a_0y+(a_0Q_y-b_0Q_x)=0 \tag{8.9}$$

如果直线用参数形式表示为 $\mathcal{L}_0(t)=P_0+t\vec{d}$，那么 $\mathcal{L}_1$ 的方程为

$$\mathcal{L}_1(t)=Q+t\vec{d}^{\perp}$$

伪码为

```
LineNormaltoLineThroughPoint(Line2D l0, Point2D q, Line2D& lOut)
{
    lOut.origin = q;
```

```
    Vector2D dPerp;
    dPerp.x = -l0.y;
    dPerp.y = l0.x;
    lOut.direction = dPerp;
}
```

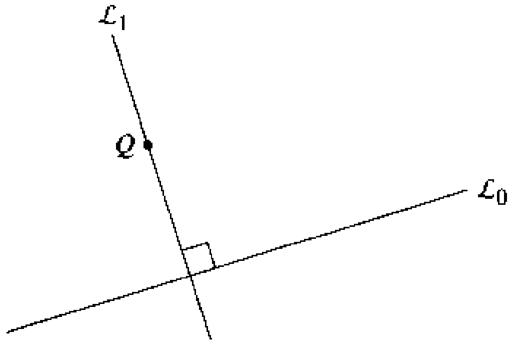


图 8.30　与一条给定直线垂直并通过一个给定点的直线

8.13　位于两点之间并与该两点等距的直线

假定我们有两个不重合的点 Q_0 和 Q_1，我们希望找到一条位于它们之间且与它们距离相等的的直线（如图 8.31 所示）。当然，任意一条经过点 Q_0 和 Q_1 的中点的直线都满足与它们距离相等的条件，但是，“位于它们之间”的意思是直线必须与向量 $Q_1 - Q_0$ 垂直；因此，问题可被看成是寻找这两个点定义的线段的垂直平分线。

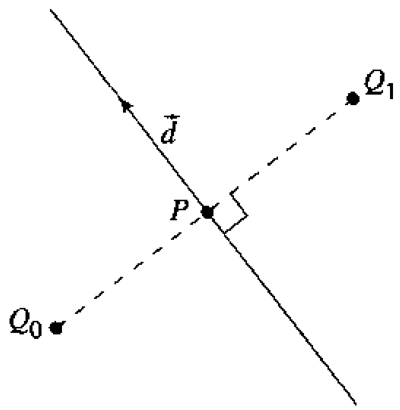


图 8.31　位于两点之间并与该两点等距的直线

参数表示要求一个点和一个方向；很清楚，点 $P = Q_0 + \frac{(Q_1 - Q_0)}{2}$ 位于直线上。由于 $\vec{d}$ 就是 $(Q_1 - Q_0)^{\perp}$，因此我们有

$$\mathcal{L}(t) = Q_0 + \frac{(Q_1 - Q_0)}{2} + t(Q_1 - Q_0)^{\perp} \tag{8.10}$$

隐含形式的计算也一样简单

$$(Q_{1,x} - Q_{0,x})x + (Q_{1,y} - Q_{0,y})y - \frac{(Q_{1,x}^2 + Q_{1,y}^2) - (Q_{0,x}^2 + Q_{0,y}^2)}{2}$$

伪码为

```
LineBetweenAndEquidistantTo2Points(Point2D q0, Point2D q1, Line2D& line)
{
    line.origin.x = q0.x + (q1.x - q0.x) / 2;
    line.origin.y = q0.y + (q1.y - q0.y) / 2;
    line.direction.x = (q0.y - q1.y);
    line.direction.y = (q1.x - q0.x);
}
```

8.14 与一条给定直线平行且相距指定值的直线

假定我们有一条直线$\mathcal{L}_0$，希望构造另一条与它平行且相距指定值d的直线$\mathcal{L}_1$，如图8.32所示。如果该直线用参数形式表示，$\mathcal{L}_0(t) = P_0 + t\vec{d}_0$，那么与其平行的直线显然具有相同的方向向量。注意到$\mathcal{L}_1$的原点必然在一条垂直于$\vec{d}_0$的直线上，很容易看出

$$P_1 = P_0 + \frac{d\vec{d}_0^{\perp}}{\|\vec{d}_0\|}$$

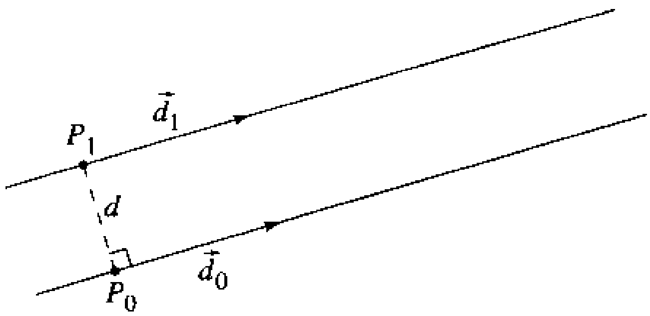


图 8.32 与一条给定直线平行且相距指定值的直线

从此可得

$$\mathcal{L}_1(t) = P_0 + \frac{d\vec{d}_0^{\perp}}{\|\vec{d}_0\|} + t\vec{d}_0$$

或

$$\mathcal{L}_1(t) = P_0 + \frac{d\vec{d}_0^{\perp}}{\|\vec{d}_0\|} + t\vec{d}_0$$

或者如果$\mathcal{L}_0$是规范的直线，我们有更简单的形式

$$\mathcal{L}_1(t) = P_0 + d\hat{d}_0^{\perp}$$

如果直线隐式地表示为$ax + by + c = 0$，那么，我们有

$$d = \begin{cases} \frac{\pm(ax+by+c)}{-\sqrt{a^2+b^2}} & b \geqslant 0 \\ \frac{\pm(ax+by+c)}{\sqrt{a^2+b^2}} & b < 0 \end{cases}$$

如果我们想使直线位于给定的直线上面，那么可以使用分子中的正号；如果我们想直线位于给定的直线下面，那么可以使用分子中的负号。如果我们代入指定的距离并做简化，就能得到目标直线的方程。

例如，如果我们有直线$\mathcal{L}: 5x - \sqrt{11}y - 7 = 0$，并且想使目标直线在它上面8个单位处，那么我们有

$$8 = \frac{5x - \sqrt{11}y - 7}{-\sqrt{5^2 + (-\sqrt{11})^2}}$$

简化后得到

$$5x - \sqrt{11}y + 39 = 0$$

伪码为

```
LineParallelToGivenLineAtGivenDistance(Line2D l1, Line2D& lOut, float distance)
{
    // Assumes l1 is not normalized
    lOut.direction = l1.direction;
    Vector2D dPerp;
    // Chose the perpendicular vector direction
    // Two answers are possible though.
    dPerp.x = -l1.direction.y;
    dPerp.y = l1.direction.x;
    float length = dPerp.length();
    lOut.origin = l1.origin + distance * dPerp / length;
}
```

8.15　与给定直线平行且垂直（水平）距离为指定值的直线

假定我们有一条给定的直线$\mathcal{L}(t) = P + t\vec{d}$，希望找到一条与$\mathcal{L}$平行且垂直距离为$v$，水平距离为$h$的直线，如图8.33所示。如果我们可以从$h$（或$v$）计算出垂直距离$d$，那么就可以简化前面所介绍的问题。利用简单的三角函数可得

$$\cos\theta = \frac{d}{v}$$

或

$$\cos\theta = \frac{d}{h}$$

分别对应于垂直和水平的情形。我们可以求解其中的d

$$v\cos\theta = d$$

或

$$h\cos\theta = d$$

分别对应于垂直和水平的情形。角度θ的余弦很容易计算

$$\cos\theta = \frac{\vec{d}^{\perp}\cdot[0\quad 1]}{\|\vec{d}\|}$$

或

$$\cos\theta = \frac{\vec{d}^{\perp}\cdot[1\quad 0]}{\|\vec{d}\|}$$

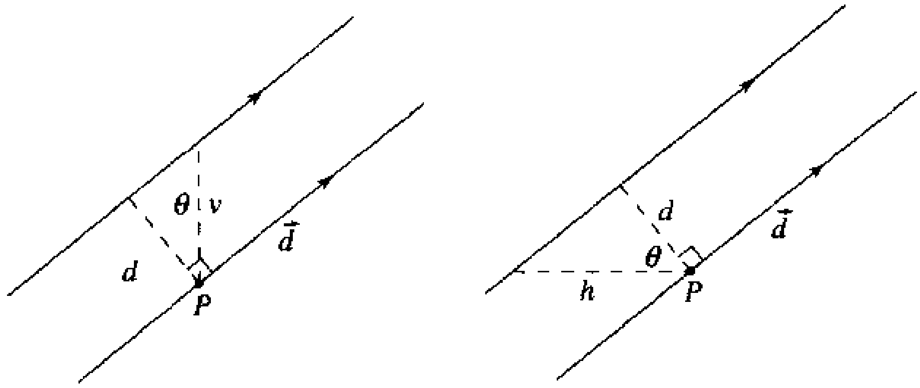


图 8.33　与一条给定直线平行且垂直或水平相距为指定值的直线

伪码为

```
LineParallelToGivenLineAtVorHDistance(Line2D l1, Line2D lOut, float distance,
                                      int vOrH)
{
    float cosTheta;
    float scalar, length;
    Vector2D dPerp;

    // Again there is another possible
    // perpendicular vector to the one chosen
    dPerp.x = -l1.y;
    dPerp.y = l1.x;

    length = l1.d.length();

    if (vOrH) {
        // vertical case
        cosTheta = dPerp.y / length;
    } else {
        // horizontal case
        cosTheta = dPerp.x / length;
    }

    scalar = distance * cosTheta;
    lOut.origin = l1.p + scalar * dPerp / length;
    lOut.direction = l1.direction;
}
```

8.16 与给定圆相切并与给定直线垂直的直线

假定我们有一个圆 C 和一条直线 $\mathcal{L}_0$，希望找到与 C 相切且垂直于 $\mathcal{L}_0$ 的直线 $\mathcal{L}_1$ 和 $\mathcal{L}_2$，如图 8.34 所示。

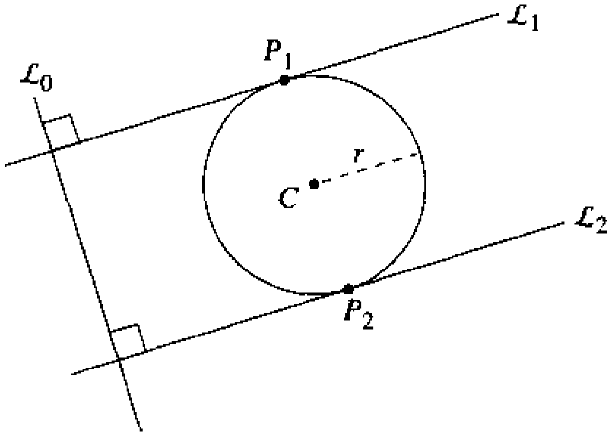


图 8.34 与一个给定圆相切并与一条给定直线垂直的直线

如果直线的方程是 $\mathcal{L}_0: ax+by+c=0$，且圆的方程为 $C:(x-x_C)+(y-y_C)-r=0$，那么目标直线的方程为

$$\mathcal{L}_0: -\frac{b}{\sqrt{a^2+b^2}}x+\frac{a}{\sqrt{a^2+b^2}}y+r+\frac{b}{\sqrt{a^2+b^2}}C_x-\frac{a}{\sqrt{a^2+b^2}}C_y=0$$

$$\mathcal{L}_1: \frac{b}{\sqrt{a^2+b^2}}x-\frac{a}{\sqrt{a^2+b^2}}y+r-\frac{b}{\sqrt{a^2+b^2}}C_x+\frac{a}{\sqrt{a^2+b^2}}C_y=0$$

伪码为

```
LinesTangentToCircleNormalToLine(Circle2D cir, Line2D l1, Line2D lOut[2])
{
    float discrm = sqrt(l1.a * l1.a + l1.b * l1.b);

    lOut[0].a = -l1.b / discrm;
    lOut[0].b =  l1.a / discrm;
    lOut[0].c =  cir.radius + ((b * cir.center.x) - (a * cir.center.y)) / discrm;

    lOut[1].a =  l1.b / discrm;
    lOut[1].b =  -l1.a / discrm;
    lOut[1].c =  cir.radius + ((-b * cir.center.x) + (a * cir.center.y)) / discrm;
}
```

如果直线具有规范的参数形式 $\mathcal{L}(t)=P_0+t\hat{d}$，那么两条新的直线就是

$$\mathcal{L}_1(t)=(C+r\hat{d})+t\hat{d}^{\perp}$$

$$\mathcal{L}_2(t)=(C-r\hat{d})+t\hat{d}^{\perp}$$

然而，如果$\mathcal{L}_0$是不规范的，那么我们有

$$\mathcal{L}_1(t)=\left(C+r\frac{\vec{d}}{\|\vec{d}\|}\right)+t\hat{d}^{\perp}$$

$$\mathcal{L}_2(t)=\left(C-r\frac{\vec{d}}{\|\vec{d}\|}\right)+t\vec{d}^{\perp}$$

伪码为

```
LinesTangentToCircleNormalToLine(Circle2D cir, Line2D l1, Line2D lOut[2])
{
    Vector2D dPerp;
    dPerp.x = -l1.direction.y;
    dPerp.y = l1.direction.x;

    if (l1.isNormalized()) {
        lOut[0].origin.x = cir.center.x + cir.radius * l1.direction.x;
        lOut[0].origin.y = cir.center.y + cir.radius * l1.direction.y;
        lOut[0].direction.x = dPerp.x;
        lOut[0].direction.y = dPerp.y;
        lOut[1].origin.x = cir.center.x - cir.radius * l1.direction.x;
        lOut[1].origin.y = cir.center.y - cir.radius * l1.direction.y;
        lOut[1].direction.x = dPerp.x;
        lOut[1].direction.y = dPerp.y;
    } else {
        float invLength = 1.0/l1.direction.length();
        lOut[0].origin.x = cir.center.x + cir.radius * l1.direction.x * invLength;
        lOut[0].origin.y = cir.center.y + cir.radius * l1.direction.y * invLength;
        lOut[0].direction.x = dPerp.x;
        lOut[0].direction.y = dPerp.y;

        lOut[1].origin.x = cir.center.x - cir.radius * l1.direction.x * invLength;
        lOut[1].origin.y = cir.center.y - cir.radius * l1.direction.y * invLength;
        lOut[1].direction.x = dPerp.x;
        lOut[1].direction.y = dPerp.y;
    }
}
```

第 9 章　三维几何图元

本章介绍在应用中常见的几种三维几何图元的定义。有的图元具有多种表示形式。在与对象相关的几何操作中，有的表示法比其他的表示法更加有效。在关于操作的讨论中，我们将说明哪种表示法更合适。

在对象多面体这类对象的几何操作中，对象可被看成是二维对象或三维对象。例如，作为二维对象，四面体就是四个三角形面的集合。作为三维对象，四面体表示其面和它所包围的区域。有的对象在使用不同的名称时就有不同的意义。例如，球面表示二维曲面，而球表示球面和其所包围的区域。如果必要，这种区别将被清楚地标识出来。在没有清楚的名称时，我们将使用立体一词。例如，在计算点与四面体之间的距离时，我们将四面体看成一个立体。如果点在四面体内，那么该距离为零。

9.1　线形对象

三维空间中最简单的直线形式是参数形式，$X(t)=P+t\vec{d}$（$t\in\mathbb{R}$），P 为直线上的一个点，且 $\vec{d}\neq\vec{0}$ 为直线的一个方向。射线是对其参数形式具有约束条件 $t\geqslant 0$ 的直线。射线的原点为 P。直线线段，或简称线段，是对其参数形式具有约束条件 $t\in[t_0,t_1]$ 的直线。如果 P_0 和 P_1 为线段的端点，线段的标准形式为 $X(t)=(1-t)P_0+tP_1$（$t\in[0,1]$）。通过设 $\vec{d}=P_1-P_0$，这种形式可以很方便地转化为参数形式。线段的对称形式包含一个中点 C，一个单位方向向量 $\hat{d}$ 和一个半径 r。其参数表示为 $X(t)=C+t\hat{d}$（$|t|\leqslant r$）。对于标准形式，线段的长度为 $\|P_1-P_0\|$；对于对称形式，其长度为 $2r$。

二维空间中的直线可等价定义为满足代数方程 $\vec{n}\cdot X=c$（$\vec{n}\neq\vec{0}$ 为直线的法线向量）的点集。直线在三维空间中的几何模拟就是两个代数方程 $\vec{n}_0\cdot X=c_0$ 和 $\vec{n}_1\cdot X=c_1$（$\vec{n}_0$ 和 $\vec{n}_1$ 线性无关）的交集。这两个线性方程具有三个未知数，即 X 的三个分量，因此我们知道，它的解中包含一个自由变量。这个变量对应于直线的参数形式中的参数。用两个平面的交集来表示直线的形式叫做直线的法线形式。

直线的参数形式可以从其法线形式中导出。叉积 $\vec{n}_0\times\vec{n}_1$ 同时垂直于线性无关向量 $\vec{n}_0$ 和 $\vec{n}_1$，因此，这三个向量构成一个线性无关集。任何一个点都可以表示为这三个向量的线性组合。特别地，$P=d_0\vec{n}_0+d_1\vec{n}_1+t\vec{n}_0\times\vec{n}_1$。定义 $e_{ij}=\vec{n}_i\cdot\vec{n}_j$。对 P 的方程两边与法线向量进行点积，我们可以得到两个方程 $c_0=\vec{n}_0\cdot P=e_{00}d_0+e_{01}d_1$ 和 $c_1=\vec{n}_1\cdot P=e_{01}d_0+e_{11}d_1$。这两个关于未知数 d_0 和 d_1 的方程可以用普通方法来求解。直线的参数形式为

$$X(t)=\frac{e_{11}c_0-e_{01}c_1}{e_{00}e_{11}-e_{01}^2}\vec{n}_0+\frac{e_{00}c_1-e_{01}c_0}{e_{00}e_{11}-e_{01}^2}\vec{n}_1+t\vec{n}_0\times\vec{n}_1$$

在本书中，线形对象用来表示直线、射线或线段。

9.2 平面对象

本节提供平面的不同定义。在许多应用中，都是在三维环境中操作标准的二维对象。这些对象必须在三维坐标系中操作，即使如此，它们依然定义在一个二维坐标系中。本节将描述建立平面二维对象的三维表示的过程。平面和任何定义在平面上的对象都统称为平面对象。

9.2.1 平面

平面用代数方程$\vec{n}\cdot(X-P)=0$（$\vec{n}\neq\vec{0}$为平面的法线向量，P为平面上的一个点，如图 9.1 所示）来定义。这种形式叫做法线－点形式。一种相似的定义为$\vec{n}\cdot X=c$（c为常数）。这种形式叫做法线－常数形式。为了用法线－常数形式在平面上建立一个点，选择$P=d\vec{n}$（d为常数）。将其代入公式，得到$c=\vec{n}\cdot(d\vec{n})=d\|\vec{n}\|^2$。另一方面，对于法线－点形式，常数$c$在法线－常数形式中为$c=\vec{n}\cdot P$。

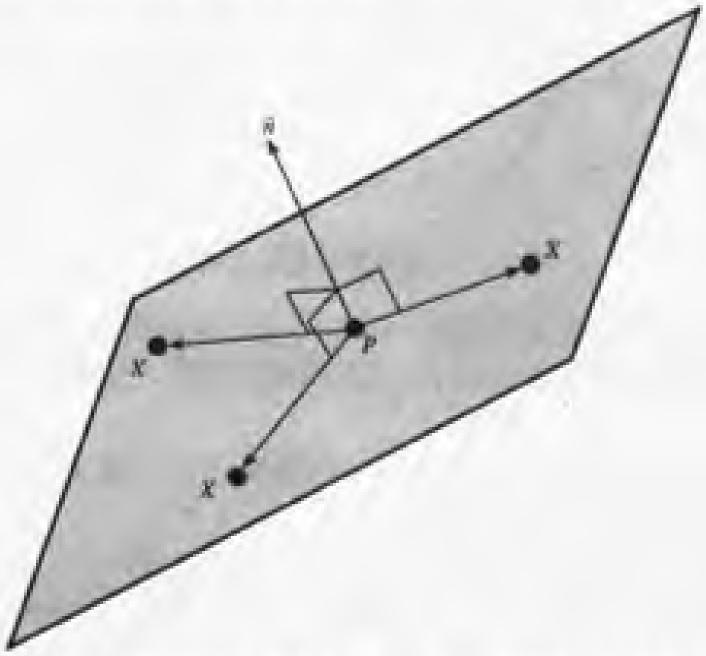


图 9.1 将平面定义为满足$\vec{n}\cdot(X-P)=0$的所有点X的集合

如果我们设$X=[x\quad y\quad z]$，可以用其分量来改写向量$X-P$，可得

$$X-P=[x-P_x\quad y-P_y\quad z-P_z]$$

如果我们设$\vec{n}=[a\quad b\quad c]$，那么我们可以将平面方程的法线－点形式改写为

$$ax+by+cz+d=0 \tag{9.1}$$

其中a,b和c为不全为零的常数，且$d=-\vec{n}\cdot P$。这叫做平面方程的隐含形式，它与法线－常数形式稍有不同，但经常在文献中遇到这种方式。

如果$a^2+b^2+c^2=1$（或者，与之等价地，如果$\|\vec{n}\|=1$），那么就称平面方程是规范的。一个非规范的表示可以通过将方程的系数乘以下式而被规范化

$$\frac{1}{\sqrt{a^2+b^2+c^2}}$$

抽象地说，使用规范表示并不是必需的，但是在许多与平面相关的算法中，如果使用规范表示，可以减小计算量。这是因为，使用规范表示时，只需在“前台”做一次平方根和除法运算，以后就可以在各种相交或距离运算中避免这类运算。

规范形式提供了对系数更直观的几何解释。观察图 9.2，我们可以看到一个垂直于页面的“平面交叉区”。应用简单的三角几何可得

$$a=\cos\alpha$$

$$b=\cos\beta$$

$$c=\cos\theta$$

其中θ为与z轴正向的夹角。

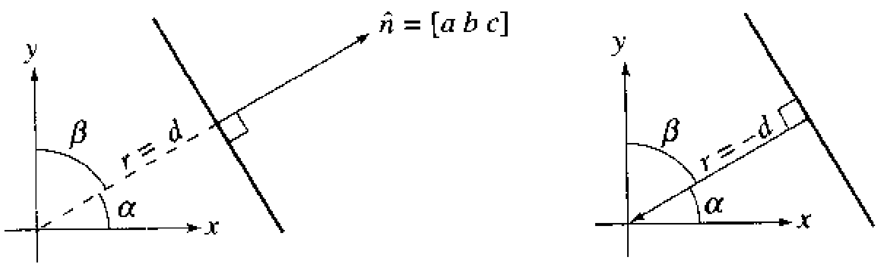


图 9.2　平面方程系数的几何意义

（至少在直观上）更值得注意的是如下几点：如果从原点到平面的距离为r，那么$|d|=r$；而且，如果$\hat{n}$从原点指向离开原点的方向，则d的符号为负，如果$\hat{n}$指向原点，则d的符号为正。

平面的参数形式为$X(s,t)=P+s\hat{u}+t\hat{v}$（$s\in\mathbb{R}$且$t\in\mathbb{R}$）。点$P$在平面上。方向$\hat{u}\neq\vec{0}$和$\hat{v}\neq\vec{0}$必须为线性无关的向量（如图 9.3 所示）。

只需将P看成是平面上的点，就可将参数形式转化为法线一点形式。法线向量必须同时垂直于两个方向向量，因此$\hat{n}=\hat{u}\times\hat{v}$。将法线一点形式转化为参数形式同样可以利用点$P$来实现。我们必须选择两个垂直于$\hat{n}$的线性无关的向量$\hat{u}$和$\hat{v}$。存在无数种选择，在这里提供一种具有健壮的数值实现的方法。基本思想是选择一个垂直于$\hat{n}=(n_0,n_1,n_2)$的单位长度向量$\hat{u}=(u_0,u_1,u_2)$，使得$\hat{u}$有一个零分量。不能确切地将任何分量选取为零分量。例如，如果你选择$\hat{u}=(u_0,u_1,0)$，那么$0=\vec{n}\cdot\hat{u}=n_0u_0+n_1u_1$。其形式化解为$\hat{u}=(n_1,-n_0,0)/\sqrt{n_0^2+n_1^2}$，然而，很清楚，当$n_0=n_1=0$时，该式存在一个代数问题；而且，当$n_0$和$n_1$都接近于 0 时，该式存在一个数值问题。因此，基于$\hat{n}$的信息来选择$\hat{u}$的为零的分量更合适。

伪码为

```
Vector N = nonzero plane normal;
Vector U, V;
```

```
if (|N.x| >= |N.y|) {
    // N.x or N.z is the largest magnitude component, swap them
    U.x = +N.z;
    U.y = 0;
    U.z = -N.x;
} else {
    // N.y or N.z is the largest magnitude component, swap them
    U.x = 0;
    U.y = +N.z;
    U.z = -N.y;
}

V = Cross(N, U);
```

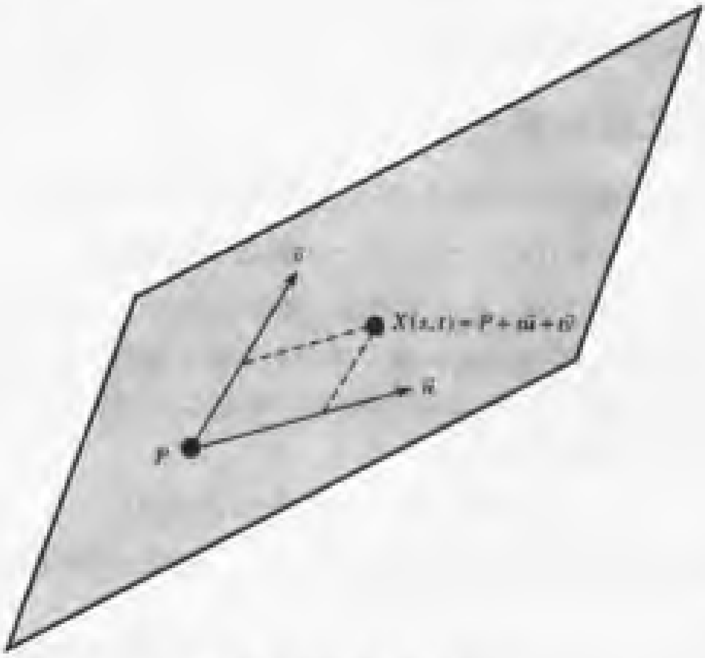


图 9.3　平面的参数表示法

9.2.2　相对于一个平面的坐标系统

对于用法线 $\vec{n}$ 和点 P 给定的一个平面，确定一个以 P 为原点、$\vec{n}$ 为一个坐标轴方向、其余两个坐标轴在平面上、完整的在 $\mathbb{R}^3$ 上的正交坐标系，有时这是很方便的。此时，$\vec{n}$ 首先被规范化为一个单位长度向量。上一节中的伪码所创建的向量 $\hat{u}$ 也被规整化为一个单位长度向量。叉积 $\hat{v} = \hat{n} \times \hat{u}$ 自动成为一个单位长度向量。

伪码为

```
Vector N = unit-length plane normal;
Vector U, V;

if (|N.x| >= |N.y|) {
    // N.x or N.z is the largest magnitude component
    invLength = 1 / sqrt(N.x * N.x + N.z * N.z);
    U.x = +N.z * invLength;
```

```
        U.y = 0;
        U.z = -N.x * invLength;
    } else {
        // N.y or N.z is the largest magnitude component
        invLength = 1 / sqrt(N.y * N.y + N.z * N.z);
        U.x = 0;
        U.y = +N.z * invLength;
        U.z = -N.y * invLength;
    }

    V = Cross(N, U);  // automatically unit length
```

任何点 $X \in \mathbb{R}^3$ 都可在上述坐标系中表示出来

$$X = P + y_0\hat{u} + y_1\hat{v} + y_2\hat{n} = P + \mathbf{R}\vec{y}$$

其中R为各列依次为 $\hat{u}, \hat{v}$ 和 $\hat{n}$ 的旋转矩阵，$\vec{y} = (y_0, y_1, y_2)$ 为 3×1 向量。

9.2.3 平面上的二维对象

考虑表示二维空间对象的 xy 平面上的一个集合 $S \subset \mathbb{R}^2$。抽象地说，

$$S = \{(x, y) \in \mathbb{R}^2 : (x, y) \text{ 满足某些约束 }\}$$

该对象可嵌入三维空间中的一个二维平面。设该平面包含一个点 P 并具有一个单位长度的法向量 $\hat{n}$。如果 $\hat{u}$ 和 $\hat{v}$ 是平面内的向量，并且 $\hat{u}, \hat{v}$ 和 $\hat{n}$ 构成一个正交集，那么二维空间内对象的（x, y）数对可以用来作为 $\hat{u}$ 和 $\hat{v}$ 的坐标，以将二维对象嵌入三维空间的平面，嵌入的集合标识为 $S' \subset \mathbb{R}^3$。该集合定义为

$$S' = \{P + x\hat{u} + y\hat{v} \in \mathbb{R}^3 : (x, y) \in S\}$$

注意，存在无数的平面，可将二维对象嵌入其中。对每一个平面又可以用无数种方法来选择向量 $\hat{u}$ 和 $\hat{v}$。

在许多应用程序中，有的问题需要用到相反的过程，即将三维空间中的一个指定平面内的对象转化为一个 xy 平面内的全等的对象。“全等”是指对一个对象通过进行刚体运动就可以转化为另一个对象。如果 S' 为平面 $\hat{n} \cdot (X - P) = 0$ 内的一个点集，通过求解 $Q = P + x\hat{u} + y\hat{v}$ 中的 x 和 y，可将任何点 $Q \in S'$ 转化为 xy 平面内的一个点。假定 $\{\hat{u}, \hat{v}, \hat{n}\}$ 为一个正交集。其解很简单：$x = \hat{u} \cdot (Q - P)$，$y = \hat{v} \cdot (Q - P)$。由于 Q 是平面上的一个点，因此 $\hat{n} \cdot (Q - P) = 0$。为了保证这两个三角形是全等的，这三个方程可以改写成如下的向量形式

$$\begin{bmatrix} x \\ y \\ 0 \end{bmatrix} = \begin{bmatrix} \hat{u} \cdot (Q - P) \\ \hat{v} \cdot (Q - P) \\ \hat{n} \cdot (Q - P) \end{bmatrix} = R(Q - P)$$

其中 R 为各列依次为 $\hat{u}$，$\hat{v}$ 和 $\hat{n}$ 的旋转矩阵。因此，xy 平面上的点 $(x, y, 0)$ 是通过先将平面 $\hat{n} \cdot (X - P) = 0$ 上的点平移，然后旋转而得到的，这里所做的是刚体运动。

【实例】 给定一个顶点为 (x_i, y_i)（$i = 0, 1, 2$）的二维三角形，并给定一个平面 $\hat{n} \cdot (X - P) = 0$，三角形就位于这个平面内，在三维空间上为三角形选择顶点的一种简单方法是 $V_i = P + x_i\hat{u} + y_i\hat{v}$（$i = 0, 1, 2$）。给定顶点 W_i（$i = 0, 1, 2$），在 xy 平面上构建一个与

原三角形全等的三角形。定义平面的原点为$P = W_0$。定义边的向量为$\vec{e}_0 = W_1 - W_0$和$\vec{e}_1 = W_2 - W_0$。三角形所在平面的单位长度法线向量为$\hat{n} = (\vec{e}_0 \times \vec{e}_1)/\|\vec{e}_0 \times \vec{e}_1\|$。用前面介绍的方法建立$\hat{u}$和$\hat{v}$。确定表达式$\vec{e}_0 = d_{00}\hat{u} + d_{01}\hat{v}$ 和$\vec{e}_1 = d_{10}\hat{u} + d_{11}\hat{v}$中的$d_{ij}$。用点积可以很简单地计算出系数，即$d_{00} = \vec{e}_0 \cdot \hat{u}$，$d_{01} = \vec{e}_0 \cdot \hat{v}$，$d_{10} = \vec{e}_1 \cdot \hat{u}$和$d_{11} = \vec{e}_1 \cdot \hat{v}$。通过这些表达式可得$W_1 = W_0 + \vec{e}_0 = P + d_{00}\hat{u} + d_{01}\hat{v}$和$W_2 = W_1 + \vec{e}_1 = P + d_{10}\hat{u} + d_{11}\hat{v}$。作为二维对象的三角形的三个顶点为（0，0)，($d_{00}, d_{01}$）和（$d_{10}, d_{11}$)，分别对应于$W_0$, W_1和W_2。■

【实例】假定你需要一个表示圆心为$C \in \mathbb{R}^3$，半径为r的圆的公式。包含该圆的平面具有单位长度法线$\hat{n}$。圆心必须在这个平面上，因此平面的方程为$\hat{n} \cdot (X - C) = 0$。圆上的点$X$与圆心$C$的距离必须都相等，因此另一个限制条件为$\|X - C\| = r$。在完整的$\mathbb{R}^3$上定义的一个代数方程产生一个中心在$C$，半径为$r$的球面。然而，我们需要的仅仅是平面上的点，因此可以将圆看成是平面和球面的交集。在二维空间中，圆心位于原点，半径为r的圆可以参数化表示为$x = r\cos\theta$和$y = r\sin\theta$（$\theta \in [0, 2\pi)$)（如图 5.15 所示)。二维空间中的圆可以形式化表示为集合$S = \{(r\cos\theta, r\sin\theta) \in \mathbb{R}^2 : \theta \in [0, 2\pi)\}$。在三维空间中，嵌入在平面内的圆为集合$S' = \{C + (r\cos\theta)\hat{u} + (r\sin\theta)\hat{v} : \theta \in [0, 2\pi)\}$（如图 9.4 所示)。如果我们定义一个向量值方程$\hat{w}(\theta) = \cos\theta\hat{u} + \sin\theta\hat{v}$，那么圆的三维参数化定义可以更加简洁地表示为

$$P = C + r\hat{w}(\theta)$$

检验$X = C + (r\cos\theta)\hat{u} + (r\sin\theta)\hat{v}$上的限制条件是很简单的。首先，

$$\begin{aligned}\hat{n} \cdot (X - C) &= \hat{n} \cdot ((r\cos\theta)\hat{u} + (r\sin\theta)\hat{v}) \\ &= (r\cos\theta)\hat{n} \cdot \hat{u} + (r\sin\theta)\hat{n} \cdot \hat{v} \\ &= (r\cos\theta)0 + (r\sin\theta)0, \quad \hat{n}\text{与}\hat{u}\text{和}\hat{v}\text{垂直}\end{aligned}$$

其次

$$\begin{aligned}\|X - C\|^2 &= \|(r\cos\theta)\hat{u} + (r\sin\theta)\hat{v}\|^2 \\ &= (r^2\cos^2\theta)\|\hat{u}\|^2 + (2r^2\sin\theta\cos\theta)\hat{u} \cdot \hat{v} + (r^2\sin^2\theta)\|\hat{v}\|^2 \\ &= (r^2\cos^2\theta)1 + (2r^2\sin\theta\cos\theta)0 + (r^2\sin^2\theta)1, \\ &\quad \hat{u}\text{和}\hat{v}\text{互相垂直} \\ &= r^2\cos^2\theta + r^2\sin^2\theta \\ &= r^2\end{aligned}$$

相似的建立方法可以应用于平面内的其他二次曲线。■

下面介绍一种获取三维空间中的平面对象S'的二维空间表示的有效方法。如果平面的法线为$\hat{n} = (n_0, n_1, n_2)$并且$n_2 \neq 0$，那么点$Q = (q_0, q_1, q_2) \in S'$在$xy$平面上的投影为$Q' = (q_0, q_1)$。条件$n_2 \neq 0$非常重要。如果它为零，那么对象所在的平面投影到一条直线上，因而将丢失原对象的很多信息。实际上，如果$n_2 = 0$，$n_1 \neq 0$，则可投影到xz平面上，如果$n_0 \neq 0$，可投影到yz平面上。在实际运用中，最重要的法线分量用来标识投影的坐标平面。与在其他坐标平面上的投影相比，这样得到的投影对象将变得尽可能地大。这种方法的一种典型应

用是三维空间中平面多边形的三角剖分。全等的映射要求对所有的多边形顶点Q进行如下的计算：$(x, y) = (\hat{u} \cdot (Q - P), \hat{v} \cdot (Q - P))$。差（$Q - P$）要求进行三次减法，并且每一次点积都要求两次乘法和一次加法。对于n个顶点的多边形，总共的运算次数为$9n$。投影映射要求标识一个非零的法线分量并且投影坐标平面的两个分量。寻找法线分量是惟一的计算开销，还需要一些浮点比较，很显然这比全等映射要节省得多。在xy平面上的三角剖分产生三个表示三角形的顶点索引。这些三元组与三维空间中的原多边形一样有效。另一个应用是计算三维空间中的平面多边形的面积，13.12 节提供了这种应用的更多细节。

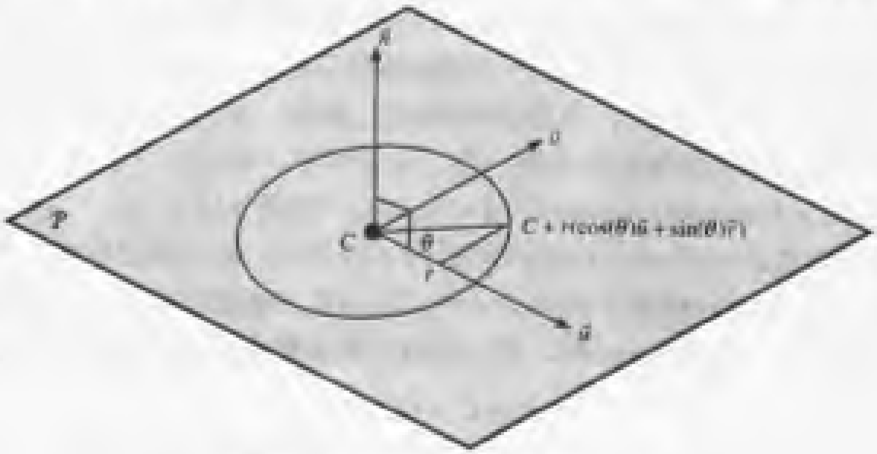


图 9.4　三维空间中圆的参数表示法

9.3　多边形网格、多面体和有限多面体

本节提供了由如下三种几何成分所构成的对象的定义：顶点、边和面。自然，顶点就是单个的点。边是端点为顶点的线段。面是三维空间中的凸多边形。许多应用程序由于存储方法太过简单，并且针对面的运算太过单调，只能支持三角形面。也可能允许出现非凸多边形面，但这只会使对象的实现和操作复杂化。如果应用程序仅要求使用三角形面，而对象的面为凸多边形，那么面可被扇形分解为三角形。如果面的n个顶点依次为V_0至V_{n-1}，那么并集为这个面的这$n-2$个三角形为$\langle V_0, V_i, V_{i+1} \rangle$（$1 \leq i \leq n-2$）。图 9.5 显示了一个凸面和将其分解为三角形的扇形分解。如果面为一个简单的非凸多边形，那么可用 13.9 节中介绍的三角剖分方法来将其分解为三角形。图 9.6 显示了一个非凸面和其三角形分解。三角剖分一般是一种费时的操作，使用网格分解的应用程序在运行时不应该使用它。因此，通常的限制是要求面本身就是三角形，或者，在最坏的情形下，是凸多边形。

满足如下条件的由有限数量的顶点、边和面组成的集合叫做多边形网格（Polygonal mesh），或者简称为多边网格（polymesh）：

- 每一个顶点必须共享至少一条边（不允许有孤立的顶点）。
- 每一条边必须共享至少一个面（不允许有孤立的边或折线）。
- 如果两个面相交，相交的顶点或边必须是网格的分量（不允许面是贯通的。一个面的边不能在另一个面的内部）。

如果所有的面都是三角形的，那么这种对象就叫做三角形网格，或简称三角网格。

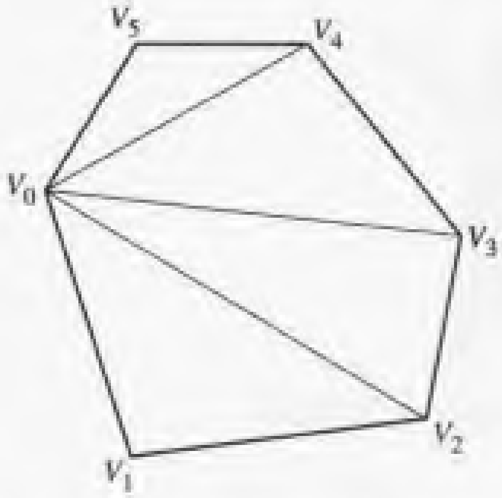


图 9.5　一个凸多边形和其三角形的扇形分解

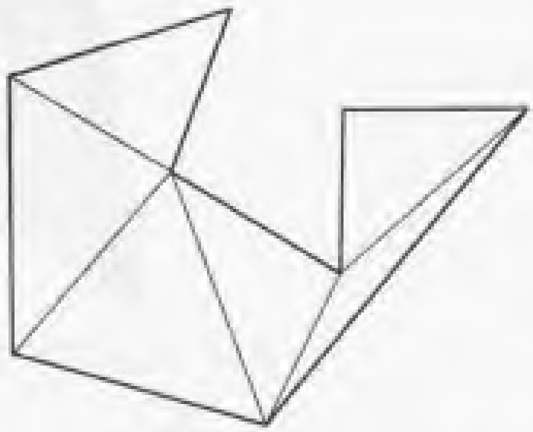


图 9.6　一个非凸多边形和其三角形的扇形分解

图 9.7 显示了一个三角网格。图 9.8 显示了一组不满足第一个条件的顶点、边和三角形，其中一个顶点是孤立的，没有用于任何一个三角形。图 9.9 显示了一组不满足第二个条件的顶点、边和三角形，其中一条边不是三角形的边，虽然它的一个端点是一个三角形的顶点。图 9.10 显示了一组不满足第三个条件的顶点、边和三角形，两个三角形是互相贯通的，它们之间的交点并不属于原来的一组点、边和三角形。



图 9.7　一个三角形网格

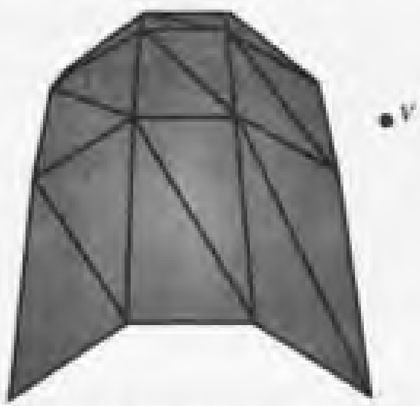


图 9.8　顶点、边和三角形不是网格，因为一个顶点是孤立的



图 9.9　顶点、边和三角形不是网格，因为一条边是孤立的

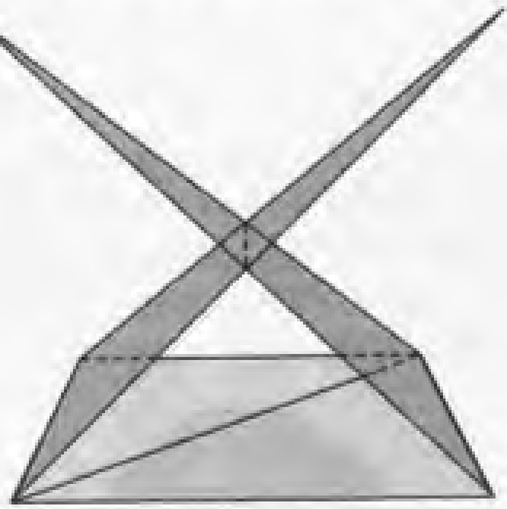


图 9.10　顶点、边和三角形不是网格，因为两个三角形互相贯通

一个多面体是具有额外约束条件的多边形网格。直观的解释是，多面体包围一个封闭的空间区域，它没有不需要的边的交接。最简单的多面体是四面体（tetrahedron），一种具有 4 个顶点、6 条边和 4 个三角形面的多边形网格。标准的四面体具有顶点$V_0 = (0, 0, 0)$，$V_1 = (1, 0, 0)$，$V_2 = (0, 1, 0)$和$V_3 = (0, 0, 1)$。其边为$E_{01} = \langle V_0, V_1\rangle$，$E_{02} = \langle V_0, V_2\rangle$，$E_{03} = \langle V_0, V_3\rangle$，$E_{12} = \langle V_1, V_2\rangle$，$E_{23} = \langle V_2, V_3\rangle$和$E_{13} = \langle V_1, V_3\rangle$。其面为$T_{012} = \langle V_0, V_1, V_2\rangle$，$T_{013} = \langle V_0, V_1, V_3\rangle$，$T_{023} = \langle V_0, V_2, V_3\rangle$和$T_{123} = \langle V_1, V_2, V_3\rangle$。将多边形网格定义为多边形的额外限制条件如下：

- 当被看成是结点为面而且弧为相邻的面所共享的边的图形时，网格是连通的。直观上，如果你能沿着从起始面到目的面的成对的相邻面的路径，从任何起始面到达目的面，那么该网格就是连通的。
- 每一条边都刚好被两个面所共享。这个条件确保网格是一个闭合并且有界的表面。

图 9.11 显示了一个多面体。图 9.12 显示了一个多边形网格，但不是一个多面体，因为它不连通。注意四面体与矩形网格共享一个顶点，但是连通性要求三角形共享边，而不仅仅是共享单一的顶点。图 9.13 显示了一个多边形网格，但不是多面体，因为有一条边被三个面共享。

凸多面体（Polytope）是封闭一个凸形区域R的多面体。即给定R内的任何两个点X和Y，线段$(1-t)X + tY$（$t \in [0, 1]$）也在R内。图 9.14 显示了一个凸多面体。

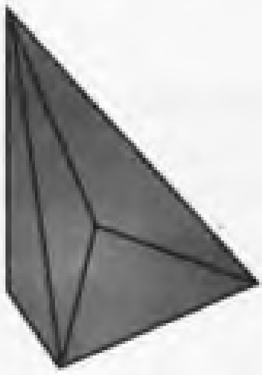


图9.11 包含一个四面体、但增加了一个多余的顶点使中心面下沉的多面体

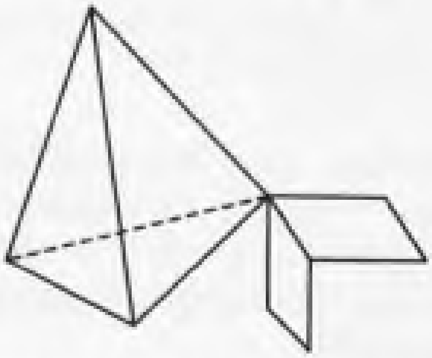


图 9.12 一个不是多面体的多边形网格，因为它没有相连在一起。四面体和矩形网格共用同一个顶点并没有使它们满足边－三角形连通性而相连在一起

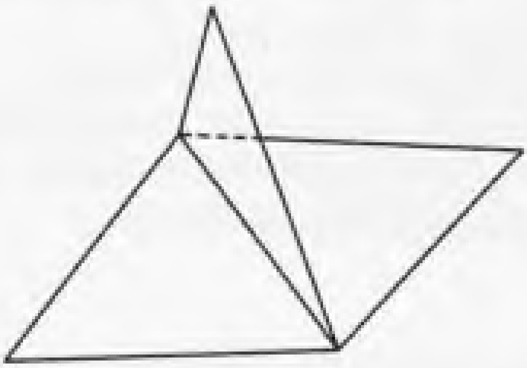


图9.13 一个不是多面体的多边形网格，因为它的三个面共用一条边

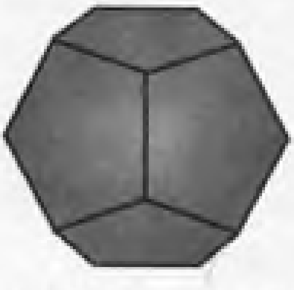


图9.14 一个凸多面体，一个规则的十二面体

9.3.1 顶点—边—面表

为了编程实现多边形网格，我们需要建立一些表示它的组分和各组分之间的邻接关系的数据结构类型。一种简单的数据结构是顶点—边—面表。N 个不同的顶点存放在一个数组中，从Vertex[0]到Vertex[N-1]，因此可通过它们的索引来引用它们在数组中的值。

边用顶点的索引对来表示，面用顶点索引的有序列表来表示。这种表可以用如下的语法来定义：

```
VertexIndex = 0 through N - 1;
VertexIndexList = EMPTY or { VertexIndex V; VertexIndexList VList; }
EdgeList = EMPTY or { Edge E; EdgeList EList; }
FaceList = EMPTY or { Face F; FaceList FList; }
Vertex = { VertexIndex V; EdgeList EList; FaceList FList; }
Edge = { VertexIndex V[2]; FaceList FList; }
Face = { VertexIndexList VList; }
```

Vertex对象中的边列表EList是有一个端点对应于索引为V的一个顶点的所有边的列表。Vertex对象中的面列表FList是有一个顶点对应于索引为V的一个顶点的所有面的列表。Edge对象中的面列表FList是共享指定边的所有面的列表。一个Edge对象并不直接知道其顶点的边共享情况。一个Face对象并不直接知道共享面的顶点或边的情况。通过查询应用于Edge或Face的子对象，可以间接获得这些信息。

从上述多边形网格的定义中可知，Edge中的面列表不能为空，因为集合中的任何边必须至少是集合中一个面的一部分。类似地，Vertex中的边和面列表都必须是非空的。如果都为空，则顶点将是孤立的。如果边列表不为空，而面列表为空，顶点将是一条孤立的折线的一部分，并且将没有面包含与其直接相邻的边。

可以根据共享它们的面的数量来对边进行分类。如果仅有一个面使用它，那么一条边是一条边界边。否则，这条边是一条内边。如果一条内边刚好有两个面共享它，它就叫做复边（manifold edge）。多面体的所有边都要求是这种类型的边。如果有三个或更多的面共享一条内边，那么这条内边就叫做连接边（junction edge）。

9.3.2 互连网格

对网格进行深度优先搜索的应用程序允许建立网格的互连组分。开始时，所有的面都标识为未被访问。从一个未被访问的面出发，该面被标识为已访问。对每一个与初始面相邻的未被访问的面进行遍历，并且对这个过程进行递归。当与初始面相邻的所有面都被遍历后，对所有面进行检查，以确定它们是否都被访问过。如果是，则该网格就是互连的。如果否，则该网格有多个互连的子网格，每一个都被称为一个互连的组分。通过对任一个未被访问的面进行递归遍历，可以发现每一个剩余的组分。下面的伪码说明了这一过程，但是其中使用的是基于堆栈的方法，而不是递归函数调用。

```
MeshList GetComponents(Mesh mesh)
{
    MeshList componentList;
```

```
    // initially all faces are unvisited
    Face f;
    for (each face f in mesh)
        f.visited = false;

    // find the connected component of an unvisited face
    while (mesh.HasUnvisitedFaces())  {
        Stack faceStack;
        f = mesh.GetUnvisitedFace();
        faceStack.Push(f);
        f.visited = true;

        // traverse the connected component of the starting face
        Mesh component;
        while (not faceStack.empty()) {
            // start at the current face
            faceStack.Pop(f);
            component.InsertFace(f);

            for (int i = 0; i < f.numEdges; i++) {
                // visit faces sharing an edge of f
                Edge e = f.edge[i];

                // visit each adjacent face
                for (int j = 0; j < e.numFaces; j++) {
                    Face a = e.face[j];
                    if (not a.visited) {

                        // this face not yet visited
                        faceStack.Push(a);
                        a.visited = true;
                    }
                }
            }
        }
        componentList.Insert(component);
    }

    return componentList;
}
```

上面的代码用于确定一个网格是否是互连的，下面的伪码对它做了一点点修改：

```
bool IsConnected(Mesh mesh)
{
    // initially all faces are unvisited
    Face f;
    for (each face f in mesh)
        f.visited = false;

    // start the traversal at any face
    Stack faceStack;
    f = mesh.GetUnvisitedFace();
```

```
        faceStack.Push(f);
        f.visited = true;

        while (not faceStack.empty()) {
            // start at the current face
            faceStack.Pop(f);
            for (int i = 0; i < f.numEdges; i++) {
                // visit faces sharing an edge of f
                Edge e = f.edge[i];

                // visit each adjacent face
                for (int j = 0; j < e.numFaces; j++) {
                    Face a = e.face[j];
                    if (not a.visited) {
                        // this face not yet visited
                        faceStack.Push(a);
                        a.visited = true;
                    }
                }
            }
        }

        // check if any face has not been visited
        for (each face f in mesh) {
            if (f.visited == false)
                return false;
        }

        // all faces were visited, the mesh is connected
        return true;
    }
```

9.3.3 复式网格

如果一个互连网格的每一条边都最多只被两个面共享，那么该网格叫做复式网格。复式网格的拓扑一般比平面内的网格更加复杂。这是一种关于可定向性的问题。虽然可定向表面有正式的数学定义，我们还是使用复式网格的定义，其中包含了进行可定向性的自我测试的含义。如果可以选取它的面的一种顶点次序，使得相邻的面具有一致的次序，就可以认为一个复式网格是可定向的。设F_0和F_1为共享边$\langle V_0, V_1\rangle$的相邻的两个面。如果V_0和V_1在面F_0上依次出现，那么，在F_1上它们出现的次序必须为V_1，V_0。最理想的情形是具有共享一条边的两个三角形的网格。图 9.15 显示了 4 种可能的构形。

麦比乌斯带是非可定向表面的一个例子。图 9.16 显示了这一点。三维空间中的一个矩形的两条平行的边可连接在一起构成一个可定向的圆柱带。可是，如果矩形被扭曲，使它的边反向连接在一起，就得到了一个麦比乌斯带，这是一个非可定向表面。

在所有的图形学应用程序中，几乎都要求网格是可定向的。请注意，在复式网格的定义中，在拓扑层面上，仅仅提及了顶点次序和互连性信息，而未涉及顶点、边和面的位置。这种定义并没有排除自我相交这类几何性质。一般的应用程序也要求网格是非自我相交的。

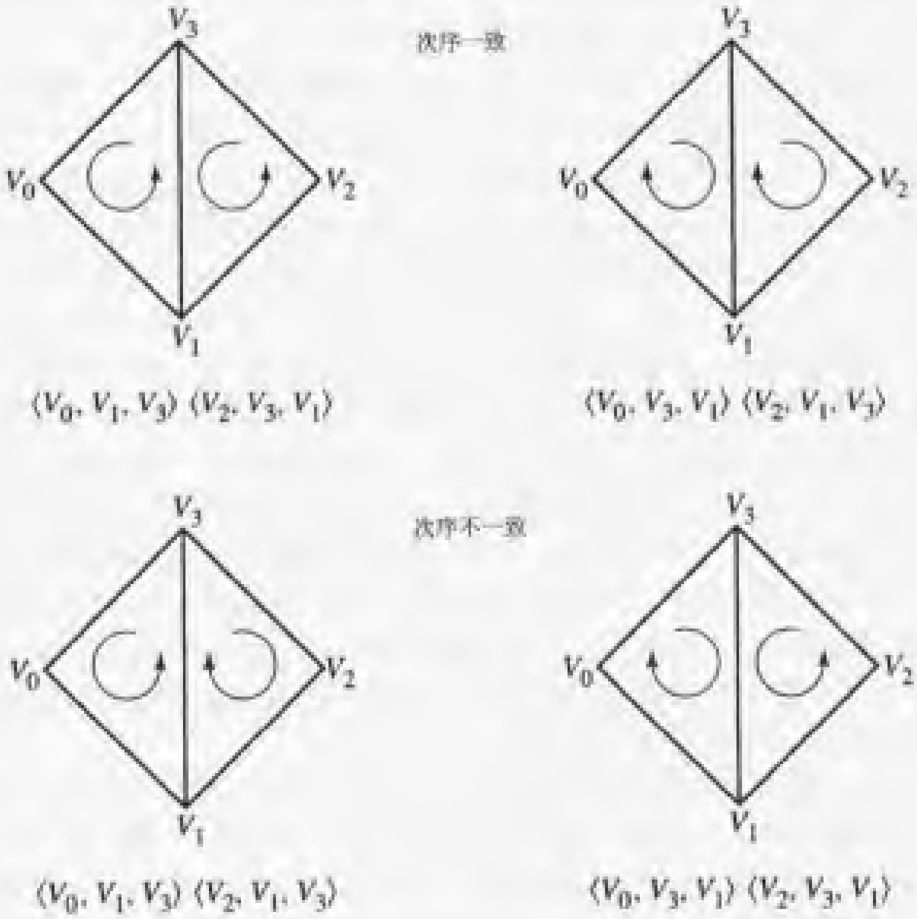


图 9.15　两个相邻三角形的顺序的 4 种可能构形



图 9.16　两条平行的边相连在一起的矩形形成了（a）一个圆柱带（可定向的）或（b）麦比乌斯带（不可定向的）

9.3.4　闭合网格

如果一个互连网格是一个复式网格，每一条边都严格地被两个面共享并且是非自我相交的，那么它被称为封闭网格。封闭网格的典型例子是镶嵌球面的三角形网格。如果一个网格不是封闭的，它就被称为开放的网格。例如，镶嵌一个半球面的三角形网格就是开放的。

9.3.5　一致次序

复式网格的可定向性条件对计算机图形学应用来说是非常重要的。一个复式网格隐含地具有关于它的面的两种一致次序。每一种次序都提供了一组面的法线。在图 9.15 中的两

个三角形的例子中，显示了一个开放的网格，上面的图形显示了网格的两种一致次序。左图的次序的法线向量是$\vec{n}_0=(V_1-V_0)\times(V_3-V_0)$和$\vec{n}_1=(V_3-V_2)\times(V_1-V_2)$。两组法线都指向图形所在的页面平面之外。右图的次序的法线向量就是$-\vec{n}_0$和$-\vec{n}_1$。一般地，一个网格的一种一致次序的法线向量集可从另一种一致次序的法线向量集导出，将所有向量取反即可。

一个应用程序必须决定对网格采用两种一致次序的哪一种。典型的选择是基于从一个观察点（相机位置）得到的网格的可见性进行确定。选择的次序将使可见的网格的面（由于参数的原因，忽略自我闭塞的情形）具有直接指向观察点的法线向量。即，如果网格的一个面在平面$\vec{n}\cdot(X-P)=0$内，且观察点为E，那么当$\vec{n}\cdot(E-P)>0$时，该面是可见的。这样的面被称为朝前的面。从观察点观察时，这个面的顶点在平面上是逆时针排序的。满足$\vec{n}\cdot(E-P)\leqslant 0$的面被称为朝后的面。在标准的图形着色系统中，可以立即舍弃朝后的面，不需要对它们进行变换或光照，这样可以节省大量的着色时间。对于存在大量封闭网格的场景，由直觉可知有将近一半的面是朝后的面。不需要画它们将极大地提高性能。

使用相同的方法来选择一致次序，一个封闭网格的法线指向网格所包围的区域的外面。正如我们在前面所提及的，这对可见性和光照来说是很重要的，而且这对几何查询来说也是很重要的。例如，一个点在凸多面体内的查询可能取决于所有法线都是指向外面这一事实。当然，如果所有法线都是指向内部的，测试依然能成功地实现。重要的是，选择一致次序，而且操作网格的不同系统也要采用这种次序。

为了实现这一目的，有时应用程序能够建立可定向的互连的复式网格，但是面的次序并不一致。经典的例子是，从三维像素数据集中抽取一个等位面，将它作为一个三角形网格。每一个三维像素都独立于其他像素而被单独处理，并且建立一组近似于等位面的三角形。这些三角形被指定为等位面上的顶点数组的索引三元组。所有三角形的集合构成了一个可定向网格（等位面总是可定向的），但是由于处理是独立进行的，三角形的次序可能是不一致的。对一些索引三元组进行重新排序以产生网格的一致次序是值得的。如果网格存储在顶点－边－面表中，对该表所表示的抽象图形进行深度优先搜索，就可以获得这种一致性。选择一个初始面。重排序是纯粹的拓扑问题，仅与顶点的次序有关，而与面的任何几何性质无关。这样，就可以获得一致次序，但是对于封闭网格，当你需要所有法线都指向外部时，你可能得到的是所有的法线都指向内部。如果预先知道需要的次序，必须提供额外的信息来对初始面进行想要的排序，可能是进行一些从观察点出发的可见性测试。初始面被标识为已访问的。如果一个相邻面具有与初始面一样的一致次序，那么不需要对相邻面做任何处理。否则，次序是不一致的，相邻的三角形需要重新排序使它具有一致次序。然后这个相邻的三角形被标识为已访问的，递归搜索其它的未被访问的相邻面。伪码列举如下。

```
void MakeConsistent(Mesh mesh)
{
    // assert: mesh is a connected manifold

    // initially all faces are unvisited
    Face f;
    for (each face f in mesh)
        f.visited = false;

    // start the traversal at any face
```

```
    Stack faceStack;
    f = mesh.GetUnvisitedFace();
    faceStack.Push(f);
    f.visited = true;

    // traverse the connected component of the starting triangle
    while (not faceStack.empty()) {
        // start at the current face
        faceStack.Pop(f);
        for (int i = 0; i < f.numEdges; i++) {
            // visit faces sharing an edge of f
            Edge e = f.edge[i];
            if (f has an adjacent face a to edge e) {
                if (not a.visited) {
                    if (a.ContainsOrderedEdge(e.V(0), e.V(1))) {
                        // f and a have inconsistent orders
                        a.ReorderVertices();
                    }
                    faceStack.Push(a);
                    a.visited = true;
                }
            }
        }
    }
}
```

9.3.6 柏拉图立体

正多边形（regular polygon）是所有边长相等并且所有边的交角都相等的凸多边形。对于顶点数为指定的$n\geqslant 3$的内接于一个单位圆的正多边形，可被描述为$(x_k, y_k) = (\cos(2\pi k/n), \sin(2\pi k/n))$（$0\leqslant k\leqslant n$）。正多面体是所有面都是全等的正多边形，并且共享每一个顶点的面的数量都相同的凸多面体。由正多面体的定义可以证明，仅有 5 种多面体可能是正多面体，但这里没有给出其证明。这些多面体叫做柏拉图立体（Platonic solid）。5 种柏拉图立体分别为四面体（tetrahedron）、六面体（hexahedron）、八面体（octahedron）、十二面体（dodecahedron）和二十面体（icosahedron）。图 9.17 显示了这 5 种立体。后面给出了不同的柏拉图立体的各种数量之间的关系。可以建立内接于一个单位球的每一种立体的顶点一面表。对于对立体着色或者提供一个为了实现镶嵌球面的目的而被分解的初始多面体来说，这个表是非常有用的。

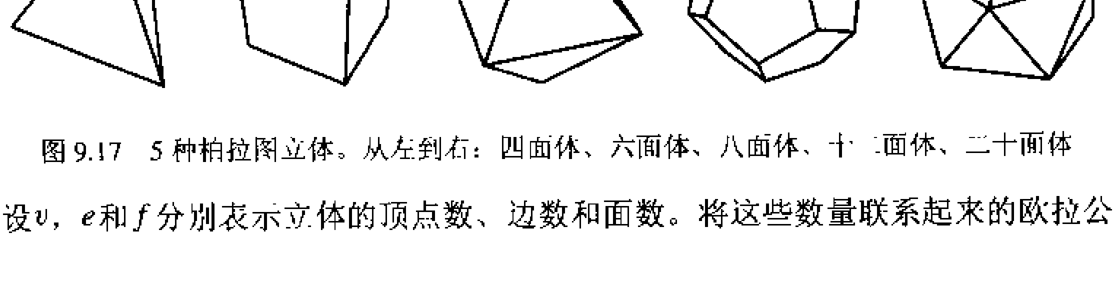


图 9.17　5 种柏拉图立体。从左到右：四面体、六面体、八面体、十二面体、二十面体

设v，e和f分别表示立体的顶点数、边数和面数。将这些数量联系起来的欧拉公式为

$v-e+f=2$。设p表示一个面的边数，q表示每一个顶点的边数。所有边的边长用L来表示。两个相邻的面之间的夹角称为两面角，用A来表示。其外接球的半径用R来表示，其内切球的半径用r来表示。其表面积用S来表示，其体积用V来表示。所有这些数量之间的关系可以用如下的方程来描述

$$\sin(A/2)=\cos(\pi/q)/\sin(\pi/p) \qquad R/L=\tan(\pi/q)\tan(A/2)/2$$

$$r/L=\cot(\pi/p)\tan(A/2)/2 \qquad R/r=\tan(\pi/p)\tan(\pi/q)$$

$$S/L^2=fp\cot(\pi/p)/4 \qquad V=rS/3$$

表 9.1 概述了柏拉图立体之间的关系。

表 9.1　不同的柏拉图立体之间的关系

	v	e	f	p	q	
四面体	4	6	4	3	3	
六面体	8	12	6	4	3	
八面体	6	12	8	3	4	
十二面体	20	30	12	5	3	
二十面体	12	30	20	3	5	
	$\sin(A)$	$\cos(A)$	R/L	r/L	S/L^2	V/L^3
四面体	$\frac{\sqrt{8}}{3}$	$\frac{1}{3}$	$\frac{\sqrt{6}}{4}$	$\frac{\sqrt{6}}{12}$	$\sqrt{3}$	$\frac{\sqrt{2}}{12}$
六面体	1	0	$\frac{\sqrt{3}}{2}$	$\frac{1}{2}$	6	1
八面体	$\frac{\sqrt{8}}{3}$	$-\frac{1}{3}$	$\frac{\sqrt{2}}{2}$	$\frac{\sqrt{6}}{6}$	$2\sqrt{3}$	$\frac{\sqrt{2}}{3}$
十二面体	$\frac{2}{\sqrt{5}}$	$-\frac{1}{\sqrt{5}}$	$\frac{\sqrt{3}(\sqrt{5}+1)}{4}$	$\frac{\sqrt{250+110\sqrt{5}}}{20}$	$3\sqrt{25+10\sqrt{5}}$	$\frac{15+7\sqrt{5}}{4}$
二十面体	$\frac{2}{3}$	$-\frac{\sqrt{5}}{3}$	$\frac{\sqrt{10+2\sqrt{5}}}{4}$	$\frac{\sqrt{42+18\sqrt{5}}}{12}$	$5\sqrt{3}$	$\frac{5(3+\sqrt{5})}{12}$

下面提供了柏拉图立体的顶点一面表。多面体的中心位于原点，而且顶点都是单位长度的。面的互连性是通过顶点数组的索引列表来提供的。当从多面体的外面观察面时，面的顶点为逆时针方向排序。四面体、八面体和二十面体的面都是三角形。六面体的面为正方形，而十二面体的面为五边形。在这两种情形中，为只支持三角形面的着色方法提供了一个顶点一三角形表。

1．四面体

顶点为

$$V_0=(0,0,1) \qquad V_2=(-\sqrt{2}/3,\sqrt{6}/3,-1/3)$$

$$V_1=(2\sqrt{2}/3,0,-1/3) \qquad V_3=(-\sqrt{2}/3,-\sqrt{6}/3,-1/3)$$

三角形互连性为

$$T_0=\langle 0,1,2\rangle \qquad T_2=\langle 0,3,1\rangle$$

$$T_1=\langle 0,2,3\rangle \qquad T_3=\langle 1,3,2\rangle$$

2. 六面体

顶点为

$$V_0 = (-1, -1, -1)/\sqrt{3} \quad V_4 = (-1, -1, 1)/\sqrt{3}$$
$$V_1 = (1, -1, -1)/\sqrt{3} \quad V_5 = (1, -1, 1)/\sqrt{3}$$
$$V_2 = (1, 1, -1)/\sqrt{3} \quad V_6 = (1, 1, 1)/\sqrt{3}$$
$$V_3 = (-1, 1, -1)/\sqrt{3} \quad V_7 = (-1, 1, 1)/\sqrt{3}$$

面互连性为

$$F_0 = \langle 0, 3, 2, 1\rangle \quad F_3 = \langle 6, 5, 1, 2\rangle$$
$$F_1 = \langle 0, 1, 5, 4\rangle \quad F_4 = \langle 6, 2, 3, 7\rangle$$
$$F_2 = \langle 0, 4, 7, 3\rangle \quad F_5 = \langle 6, 7, 4, 5\rangle$$

三角形互连性为

$$T_0 = \langle 0, 3, 2\rangle \quad T_6 = \langle 6, 5, 1\rangle$$
$$T_1 = \langle 0, 2, 1\rangle \quad T_7 = \langle 6, 1, 2\rangle$$
$$T_2 = \langle 0, 1, 5\rangle \quad T_8 = \langle 6, 2, 3\rangle$$
$$T_3 = \langle 0, 5, 4\rangle \quad T_9 = \langle 6, 3, 7\rangle$$
$$T_4 = \langle 0, 4, 7\rangle \quad T_{10} = \langle 6, 7, 4\rangle$$
$$T_5 = \langle 0, 7, 3\rangle \quad T_{11} = \langle 6, 4, 5\rangle$$

3. 八面体

顶点为

$$V_0 = (1, 0, 0) \quad V_3 = (0, -1, 0)$$
$$V_1 = (-1, 0, 0) \quad V_4 = (0, 0, 1)$$
$$V_2 = (0, 1, 0) \quad V_5 = (0, 0, -1)$$

三角形互连性为

$$T_0 = \langle 4, 0, 2\rangle \quad T_4 = \langle 5, 2, 0\rangle$$
$$T_1 = \langle 4, 2, 1\rangle \quad T_5 = \langle 5, 1, 2\rangle$$
$$T_2 = \langle 4, 1, 3\rangle \quad T_6 = \langle 5, 3, 1\rangle$$
$$T_3 = \langle 4, 3, 0\rangle \quad T_7 = \langle 5, 0, 3\rangle.$$

4. 十二面体

用如下的中间项来建立顶点：$a = 1/\sqrt{3}$，$b = \sqrt{(3 - \sqrt{5})/6}$ 以及 $c = \sqrt{(3 + \sqrt{5})/6}$。顶点为

$$V_0 = (a, a, a) \qquad V_{10} = (b, -c, 0)$$
$$V_1 = (a, a, -a) \qquad V_{11} = (-b, -c, 0)$$
$$V_2 = (a, -a, a) \qquad V_{12} = (c, 0, b)$$
$$V_3 = (a, -a, -a) \qquad V_{13} = (c, 0, -b)$$
$$V_4 = (-a, a, a) \qquad V_{14} = (-c, 0, b)$$
$$V_5 = (-a, a, -a) \qquad V_{15} = (-c, 0, -b)$$
$$V_6 = (-a, -a, a) \qquad V_{16} = (0, b, c)$$
$$V_7 = (-a, -a, -a) \qquad V_{17} = (0, -b, c)$$
$$V_8 = (b, c, 0) \qquad V_{18} = (0, b, -c)$$
$$V_9 = (-b, c, 0) \qquad V_{19} = (0, -b, -c)$$

面互连性为

$$F_0 = \langle 0, 8, 9, 4, 16\rangle \qquad F_6 = \langle 0, 12, 13, 1, 8\rangle$$
$$F_1 = \langle 0, 16, 17, 2, 12\rangle \qquad F_7 = \langle 8, 1, 18, 5, 9\rangle$$
$$F_2 = \langle 12, 2, 10, 3, 13\rangle \qquad F_8 = \langle 16, 4, 14, 6, 17\rangle$$
$$F_3 = \langle 9, 5, 15, 14, 4\rangle \qquad F_9 = \langle 6, 11, 10, 2, 17\rangle$$
$$F_4 = \langle 3, 19, 18, 1, 13\rangle \qquad F_{10} = \langle 7, 15, 5, 18, 19\rangle$$
$$F_5 = \langle 7, 11, 6, 14, 15\rangle \qquad F_{11} = \langle 7, 19, 3, 10, 11\rangle$$

三角形互连性为

$$T_0 = \langle 0, 8, 9\rangle \qquad T_{12} = \langle 0, 9, 4\rangle \qquad T_{24} = \langle 0, 4, 16\rangle$$
$$T_1 = \langle 0, 12, 13\rangle \qquad T_{13} = \langle 0, 13, 1\rangle \qquad T_{25} = \langle 0, 1, 8\rangle$$
$$T_2 = \langle 0, 16, 17\rangle \qquad T_{14} = \langle 0, 17, 2\rangle \qquad T_{26} = \langle 0, 2, 12\rangle$$
$$T_3 = \langle 8, 1, 18\rangle \qquad T_{15} = \langle 8, 18, 5\rangle \qquad T_{27} = \langle 8, 5, 9\rangle$$
$$T_4 = \langle 12, 2, 10\rangle \qquad T_{16} = \langle 12, 10, 3\rangle \qquad T_{28} = \langle 12, 3, 13\rangle$$
$$T_5 = \langle 16, 4, 14\rangle \qquad T_{17} = \langle 16, 14, 6\rangle \qquad T_{29} = \langle 16, 6, 17\rangle$$
$$T_6 = \langle 9, 5, 15\rangle \qquad T_{18} = \langle 9, 15, 14\rangle \qquad T_{30} = \langle 9, 14, 4\rangle$$
$$T_7 = \langle 6, 11, 10\rangle \qquad T_{19} = \langle 6, 10, 2\rangle \qquad T_{31} = \langle 6, 2, 17\rangle$$
$$T_8 = \langle 3, 19, 18\rangle \qquad T_{20} = \langle 3, 18, 1\rangle \qquad T_{32} = \langle 3, 1, 13\rangle$$
$$T_9 = \langle 7, 15, 5\rangle \qquad T_{21} = \langle 7, 5, 18\rangle \qquad T_{33} = \langle 7, 18, 19\rangle$$
$$T_{10} = \langle 7, 11, 6\rangle \qquad T_{22} = \langle 7, 6, 14\rangle \qquad T_{34} = \langle 7, 14, 15\rangle$$
$$T_{11} = \langle 7, 19, 3\rangle \qquad T_{23} = \langle 7, 3, 10\rangle \qquad T_{35} = \langle 7, 10, 11\rangle$$

5. 二十面体

设$t = (1 + \sqrt{5})/2$。顶点为

$$V_0=(t,1,0)/\sqrt{1+t^2} \qquad V_6=(-1,0,t)/\sqrt{1+t^2}$$
$$V_1=(-t,1,0)/\sqrt{1+t^2} \qquad V_7=(-1,0,-t)/\sqrt{1+t^2}$$
$$V_2=(t,-1,0)/\sqrt{1+t^2} \qquad V_8=(0,t,1)/\sqrt{1+t^2}$$
$$V_3=(-t,-1,0)/\sqrt{1+t^2} \qquad V_9=(0,-t,1)/\sqrt{1+t^2}$$
$$V_4=(1,0,t)/\sqrt{1+t^2} \qquad V_{10}=(0,t,-1)/\sqrt{1+t^2}$$
$$V_5=(1,0,-t)/\sqrt{1+t^2} \qquad V_{11}=(0,-t,-1)/\sqrt{1+t^2}$$

三角形互连性为

$$T_0=\langle 0,8,4\rangle \qquad T_{10}=\langle 2,9,11\rangle$$
$$T_1=\langle 0,5,10\rangle \qquad T_{11}=\langle 3,11,9\rangle$$
$$T_2=\langle 2,4,9\rangle \qquad T_{12}=\langle 4,2,0\rangle$$
$$T_3=\langle 2,11,5\rangle \qquad T_{13}=\langle 5,0,2\rangle$$
$$T_4=\langle 1,6,8\rangle \qquad T_{14}=\langle 6,1,3\rangle$$
$$T_5=\langle 1,10,7\rangle \qquad T_{15}=\langle 7,3,1\rangle$$
$$T_6=\langle 3,9,6\rangle \qquad T_{16}=\langle 8,6,4\rangle$$
$$T_7=\langle 3,7,11\rangle \qquad T_{17}=\langle 9,4,6\rangle$$
$$T_8=\langle 0,10,8\rangle \qquad T_{18}=\langle 10,5,7\rangle$$
$$T_9=\langle 1,8,10\rangle \qquad T_{19}=\langle 11,7,5\rangle$$

9.4 二次曲面

Finney 和 Thomas（1996）中提供了有关二次曲面的非常精彩的讨论，但是其中的讨论将所有的方程都考虑为轴对齐的形式。本节的讨论涉及一般的二次方程，并以描述曲面的矩阵的特征值分解为基础。特征值分解算法将在 A.3 节中介绍。

一般的二次方程为$X^{\mathrm{T}}\mathbf{A}X+B^{\mathrm{T}}X+c=0$，其中A为一个 3×3 非零对称矩阵，$B$为一个 3×1 向量，且$c$是一个常数。3×1 向量$X$表示变量的数量。由于A是对称的，因此它可因式分解为$\mathbf{A}=\mathbf{R}^{\mathrm{T}}\mathbf{D}\mathbf{R}$，其中D为一个对角矩阵，其对角线元素为A的特征值，并且R是一个旋转矩阵，它的各行对应于特征向量。设$Y=\mathbf{R}X$且$E=\mathbf{R}B$，那么二次方程为$Y^{\mathrm{T}}\mathbf{D}Y+E^{\mathrm{T}}Y+c=0$。通过对各项进行平方，可以对二次方程进行因式分解。这允许我们确定曲面的类型或者确定解为退化的曲线（点，线，平面）。设$\mathbf{D}=\mathrm{Diag}(d_0,d_1,d_2)$和$E=(e_0,e_1,e_2)$。

9.4.1 三个非零特征值

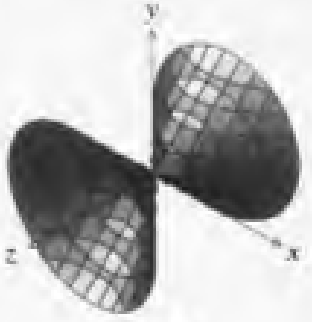
因子组合后的方程为

$$d_0\left(y_0+\frac{e_0}{2d_0}\right)^2+d_1\left(y_1+\frac{e_1}{2d_1}\right)^2+d_2\left(y_2+\frac{e_2}{2d_2}\right)^2+c-\frac{e_0^2}{4d_0}-\frac{e_1^2}{4d_1}-\frac{e_2^2}{4d_2}=0$$

定义$\gamma_i = -e_i/(2d_i)$（$i = 0, 1, 2$），并定义$f = e_0^2/4d_0 + e_1^2/4d_1 + e_2^2/4d_2 - c$。方程为$d_0(y_0 - \gamma_0)^2 + d_1(y_1 - \gamma_1)^2 + d_2(y_2 - \gamma_2)^2 = f$。

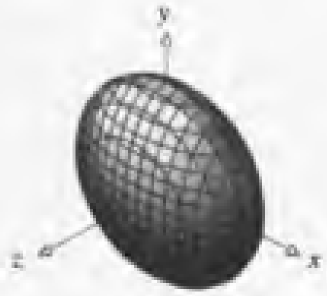
假定$f = 0$。如果所有特征值都为正或者都为负，那么方程表示一个点$(\gamma_0, \gamma_1, \gamma_2)$。如果至少一个特征值为正并且一个特征值为负，重新对各项排序，并且可能需要乘以−1，使得$d_0 > 0$，$d_1 > 0$且$d_2 < 0$。方程为$(y_2 - \gamma_2)^2 = (-d_0/d_2)(y_0 - \gamma_0)^2 + (-d_1/d_2)(y_1 - \gamma_1)^2$，表示一个椭圆锥面（elliptic cone）。

设$f > 0$；否则对方程两边乘以−1，使f为正。如果所有的特征值都为负，则方程无解。如果所有特征值都为正，那么方程表示一个椭圆体（ellipsoid）。其中心为$(\gamma_0, \gamma_1, \gamma_2)$，半轴长度为$\sqrt{f/d_i}$（$i = 0, 1, 2$）。如果至少一个特征值为正并且一个特征值为负，那么方程表示一个双曲面（hyperboloid）（一个或两个面取决于正的特征值的数量）。图 9.18 显示了这些二次曲面，以及它们的标准（轴对齐）方程。



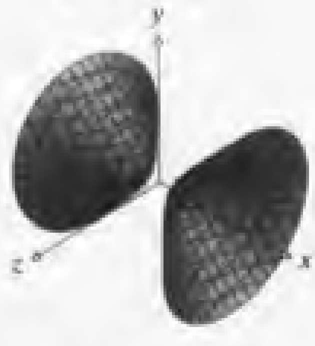
椭圆锥面

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 0$$



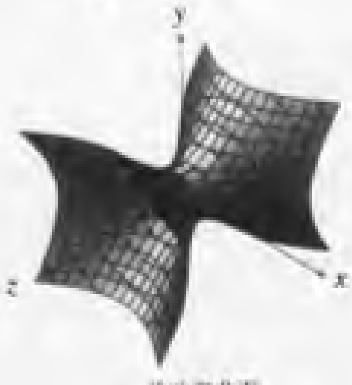
椭圆体

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$$



双叶双曲面

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = -1$$



单叶双曲面

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} - \frac{z^2}{c^2} = 1$$

图 9.18 具有三个非零特征值的二次曲面

9.4.2 两个非零特征值

不失一般性，假设$d_2 = 0$。因子组合后的方程为

$$d_0\left(y_0 + \frac{e_0}{2d_0}\right)^2 + d_1\left(y_1 + \frac{e_1}{2d_1}\right)^2 + e_2 y_2 + c - \frac{e_0^2}{4d_0} - \frac{e_1^2}{4d_1} = 0$$

定义$\gamma_i = -e_i/(2d_i)$（$i = 0,1$），并定义$f = e_0^2/4d_0 + e_1^2/4d_1 - c$。方程为$d_0(y_0 - \gamma_0)^2 + d_1(y_1 - \gamma_1)^2 + e_2 y_2 = f$。

假定$e_2 = 0$且$f = 0$。如果d_0和d_1都为正或者都为负，那么方程表示一条包含点$(\gamma_0, \gamma_1, 0)$且方向为（0，0，1）的直线。否则，特征值具有相反的符号，方程表示如下两个面的并集：$y_1 - \gamma_1 = \pm\sqrt{-d_0/d_1}(y_0 - \gamma_0)$。

设$e_2 = 0$且$f > 0$（如果$f < 0$，对方程两边乘以-1）。如果d_0和d_1都为负，则方程无解。如果都为正，那么方程表示一个椭圆柱体（elliptic cylinder）（如果$d_0 = d_1$，则为圆柱体）。否则，d_0和d_1具有相反的符号，方程表示一个双曲线柱体（hyperbolic cylinder）。

设$e_2 \neq 0$。定义$\gamma_2 = f/e_q$。方程为$d_0(y_0 - \gamma_0)^2 + d_1(y_1 - \gamma_1)^4 + e_2(y_2 - \gamma_2) = 0$。如果$d_0$和$d_1$具有相同的符号，则方程表示一个椭圆抛物面（elliptic paraboloid）（如果$d_0 = d_1$，则为圆抛物面）。否则，d_0和d_1具有相反的符号，方程表示一个双曲线抛物面（hyperbolic paraboloid）。图 9.19 显示了这些二次曲面。

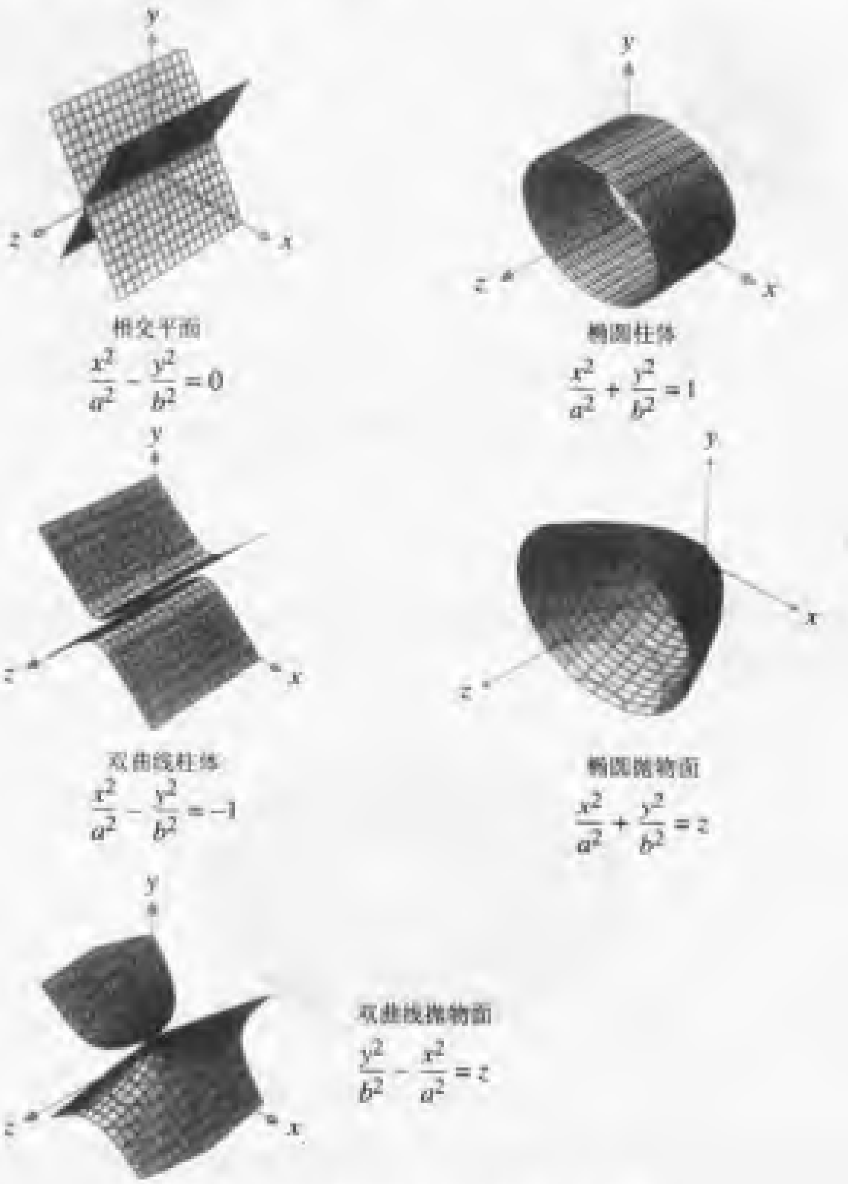


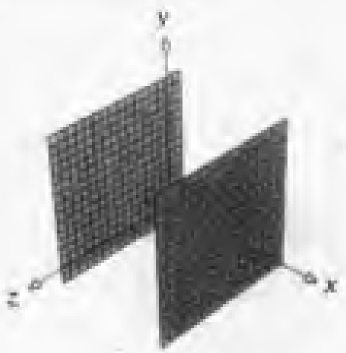
图 9.19　具有两个非零特征值的二次曲面

9.4.3 一个非零特征值

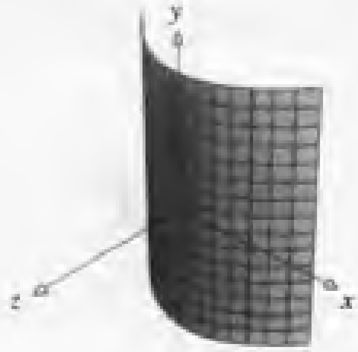
因子组合后的方程为

$$d_0\left(y_0+\frac{e_0}{2d_0}\right)^2+e_1y_1+e_2y_2+c-\frac{e_0^2}{4d_0}=0$$

如果$e_1=e_2=0$，那么方程退化（或者无解，或者如果y_0为常数，此时解为一个平面）。否则，定义$L=\sqrt{e_1^2+e_2^2}\neq 0$且对方程的两边除以$L$。定义$\alpha=d_0/L$，$\beta=(c-e_0^2/(4d_0))/L$，并对变量做刚体变换$z_0=y_0+e_0/(2d_0)$，$z_1=-(e_1y_1+e_2y_2)/L$，$z_2=(-e_2y_1+e_1y_2)/L$。在新的坐标系中的方程为$z_1=\alpha z_0^2+\beta$，因此曲面为抛物柱体（parabolic cylinder）。图 9.20 显示了这些二次曲面。



平行平面
$x^2=a^2$



抛物柱面
$x^2+2rz=0$

图 9.20　具有一个非零特征值的二次曲面

9.5 环面

三维空间中的一个环面是一个四次曲面，（在最常见的比例范围内）具有通常称为“多纳圈”的形状，如图 9.21 所示。



图 9.21　一个标准的环面

可以认为环面是用如下方法来生成的，即将圆绕其所在平面内的一条轴旋转，或者将一个矩形片的相对的边连在一起（不要扭曲）。几种其他的方法也可能得到环面：两种不

同的隐含定义为

$$x^4 + y^4 + z^4 + x^2y^2 + 2x^2z^2 + 2y^2z^2 - 2(r_0^2 + r_1^2)x^2 + 2(r_0^2 - r_1^2)y^2$$
$$- 2(r_0^2 + r_1^2)z^2 + (r_0^2 - r_1^2)^2 = 0$$

和

$$(r_0 - \sqrt{x^2 + y^2})^2 + z^2 = r_1^2$$

以及一种参数定义

$$x = (r_0 + r_1 \cos v) \cos u$$
$$y = (r_0 + r_1 \cos v) \sin u$$
$$z = r_1 \sin v$$

其中r_0为从环面的中心到“管子”的中心的半径（主半径），r_1为“管子”自己的半径（次半径）。一般地，主半径大于次半径（$r_0 > r_1$），这对应于一个环形环面。其余两种为喇叭形环面($r_0 = r_1$)和自相交轴形环面(self-intersecting spindle torus)($r_0 < r_1$)(Weisstein 1999)。

环形环面的表面积S和体积V很容易计算（Weisstein 1999)。我们知道，半径为r的圆的周长和面积分别为$2\pi r$和πr^2，而且环面可以被看成是一个圆绕与其所在的平面平行的轴旋转而得到的表面，因此可以直接得到，

$$S = (2\pi r_1)(2\pi r_0)$$
$$= 4\pi^2 r_1 r_0$$

和

$$V = (2\pi r_1^2)(2\pi r_0)$$
$$= 2\pi^2 r_1^2 r_0$$

9.6　多项式曲线

空间中的多项式曲线（polynomial curve）是向量值函数$X(t)$，即$X: D \subset \mathbb{R} \to R \subset \mathbb{R}^3$，其定义域为$D$，值域为$R$。$X(t)$的分量$X_i(t)$都是具有指定参数的多项式

$$X_i(t) = \sum_{j=0}^{n_i} a_{ij} t^j$$

其中n_i为多项式的次数。在大多数应用程序中，各个分量的次数都相同，此时曲线可表示为$X(t) = \sum_{j=0}^{n} A_j t^j$，$A_j \in \mathbb{R}^3$为已知的点。定义域一般为$\mathbb{R}$或$[0, 1]$。一条有理多项式曲线（rational polynomial curve）是向量值函数$X(t)$，其各分量$X_i(t)$都是多项式的比值

$$X_i(t) = \frac{\sum_{j=0}^{n_i} a_{ij} t^j}{\sum_{j=0}^{m_i} b_{ij} t^j}$$

其中n_i和m_i分别为分子和分母多项式的次数。

计算机图形学中常用的几种曲线类型包括贝塞尔曲线、B样条曲线和非均匀有理B样条（NURBS）曲线。这里仅给出这些曲线的定义。它们的各种有趣的性质可在其他的书籍中找到（Bartels，Beatty 和 Barsky 1987；Cohen，Riesenfeld 和 Elber 2001；Farin 1990，1995；Rogers 2001；Yamaguchi 1988）。

9.6.1 贝塞尔曲线

空间上的贝塞尔曲线可由点集$P_i \in \mathbb{R}^3$（$0 \leqslant i \leqslant n$）（这些点叫做控制点）按如下的方式构成：

$$X(t)=\sum_{i=0}^{n}\binom{n}{i}t^i(1-t)^{n-1}P_i=\sum_{i=0}^{n}B_i(t)P_i$$

其中$t\in[0,1]$。实数值多项式$B_i(t)$叫做伯恩斯坦多项式，其中每一个多项式的次数都为n。因此，$X(t)$的多项式分量的次数也为n。图 9.22 显示了一条三次贝塞尔曲线，以及控制点和控制多边形。

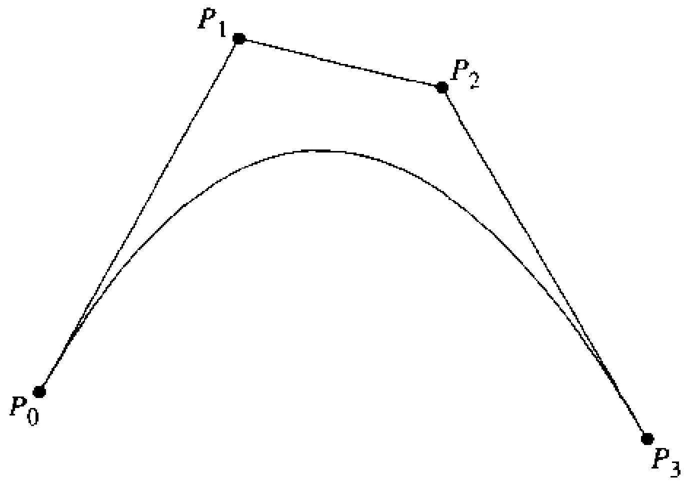


图 9.22　一条三次贝塞尔曲线

9.6.2 B 样条曲线

空间上次数为j的 B 样条曲线可以由点集$P_i \in \mathbb{R}^3$（这些点叫做控制点）和单调参数集t_i（$t_i \leqslant t_{i+1}$）（这些点叫做结点）（$0 \leqslant i \leqslant n$）按如下的方式构成：

$$X(t)=\sum_{i=0}^{n}B_{i,j}(t)P_i$$

其中$t\in[t_0,t_n]$且$1\leqslant j \leqslant n$。向量（$t_0,\ldots,t_n$）叫做结点向量（knot vector）。实数值多项式$B_{i,j}(t)$的次数为j，由 Cox-de Boor 递归公式定义

$$B_{i,0}(t)=\begin{cases}1, & t_i\leqslant t<t_{i+1}\\ 0, & 否则\end{cases}$$

$$B_{i,j}(t)=\frac{(t-t_i)B_{i,j-1}(t)}{t_{i+j-1}-t_i}+\frac{(t_{i+j}-t)B_{i+1,j-1}(t)}{t_{i+j}-t_{i+1}}$$

其中$1\leqslant j\leqslant n$。$X(t)$的多项式分量实际上是在区间$[t_i,t_{i+1}]$上分段定义的。多项式在每一个区间上的次数都为j。结点的值并不要求是均匀分布的。这种情形中的曲线叫做非均匀B样条曲线。如果结点的值是均匀分布的，曲线就叫做均匀B样条曲线。图 9.23 显示了一条三次B样条曲线，以及控制点和控制多边形。

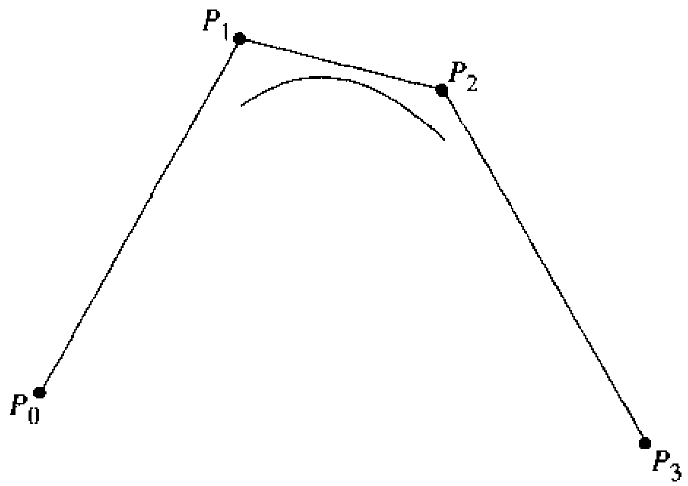


图 9.23　一条三次B样条曲线

9.6.3　非均匀有理 B 样条曲线

空间上的非均匀有理B样条曲线（或称 NURBS 曲线）是从三维空间中的非均匀B样条多项式曲线中得出的。控制点为$(P_i,1)\in\mathbb{R}^4$（$0\leqslant i\leqslant n$），多项式曲线为

$$(Y(t),w(t))=\sum_{i=0}^{n}B_{i,j}(t)w_i(P_i,1)$$

其中权重$w_i>0$，$B_{i,j}(t)$就是上一节中定义的同名多项式。NURBS 曲线可通过如下方法获得，将（$Y(t),w(t)$）作为齐次向量，并除以最后一个分量，可以得到在三维空间上的一个投影

$$X(t)=\frac{Y(t)}{w(t)}=\sum_{i=0}^{n}R_{i,j}(t)P_i$$

其中

$$R_{i,j}(t)=\frac{w_iB_{i,j}(t)}{\sum_{i=0}^{n}w_iB_{i,j}(t)}$$

9.7　多项式曲面

空间中的多项式曲面是向量值函数$X(s,t)$，即$X:D\subset\mathbb{R}^2\to\mathbb{R}^3$，其定义域为$D$，值域为$R$。$X(s,t)$的分量$X_i(s,t)$都是具有指定参数的多项式

$$X_i(s,t)=\sum_{j=0}^{n_i}\sum_{k=0}^{m_i}a_{ijk}s^jt^k$$

其中n_i+m_i为多项式的次数。定义域D一般为$\mathbb{R}^2$或$[0,1]^2$。一个有理多项式曲面是一个向量值函数$X(s,t)$，其各分量$X_i(s,t)$都是多项式的比值

$$X_i(s,t)=\frac{\sum_{j=0}^{n_i}\sum_{k=0}^{m_i}a_{ijk}s^jt^k}{\sum_{j=0}^{p_i}\sum_{k=0}^{q_i}b_{ijk}s^jt^k}$$

其中n_i+m_i为分子多项式的次数，且p_i+q_i为分母多项式的次数。

计算机图形学中常用的几种曲面类型包括贝塞尔曲面、B样条曲面和非均匀有理B样条（NURBS）曲面。这里仅给出这些曲面的定义。它们的各种有趣的性质可在其他的书籍中找到（Bartels，Beatty 和 Barsky 1987；Cohen，Riesenfeld 和 Elber 2001；Farin 1990，1995；Rogers 2001；Yamaguchi 1988）。

9.7.1 贝塞尔曲面

这里定义的两种曲面为贝塞尔矩形补面和贝塞尔三角形补面。

1. 贝塞尔矩形补面

给定一个三维空间控制点的矩形格子P_{i_0,i_1}（$0\leqslant i_0\leqslant n_0$，$0\leqslant i_1\leqslant n_1$），对这些点的贝塞尔矩形补面为

$$X(s,t)=\sum_{i_0=0}^{n_0}\sum_{i_1=0}^{n_1}B_{n_0,i_0}(s)B_{n_1,i_1}(t)\,P_{i_0,i_1},\quad (s,t)\in[0,1]^2$$

其中

$$\binom{n}{i}=\frac{n!}{i!(n-i)!}$$

是从一组 n 项中选出的 i 项的组合的数量。其中的系数由伯恩斯坦多项式产生

$$B_{n,i}(z)=\binom{n}{i}z^i(1-z)^{n-i}$$

这种补面叫做矩形补面是因为定义域$[0,1]^2$在st平面上是一个矩形。图 9.24 显示了一个双三次贝塞尔曲面，以及其控制点和控制多边形。

2. 贝塞尔三角形补面

给定一个三维空间控制点的矩形格子P_{i_0,i_1,i_2}（$i_0\geqslant 0$，$i_1\geqslant 0$，$i_2\geqslant 0$，且$i_0+i_1+i_2=n$），对这些点的贝塞尔三角形补面为

$$X(u,v,w)=\sum_{|I|=n}B_{n,I}(u,v,w)\,P_I$$

其中$I=(i_0,i_1,i_2)$，$|I|=i_0+i_1+i_2$，$u\geqslant 0$，$v\geqslant 0$，$w\geqslant 0$且$u+v+w=1$。总和涉及$(n+1)(n+2)/2$项。伯恩斯坦多项式系数为

$$B_{n,I}(u,v,w)=\binom{n}{i_0,i_1,i_2}u^{i_0}v^{i_1}w^{i_2}=\frac{n!}{i_0!i_1!i_2!}u^{i_0}v^{i_1}w^{i_2}$$



图 9.24　一个双三次B 样条曲面

虽然补面具有三个变量 u，v 和 w，然而 $w=1-u-v$ 说明 X 仅依赖于 u 和 v。这种补面之所以叫做三角形补面，是因为定义域 $u\geqslant0$，$v\geqslant0$，$w\geqslant0$ 且 $u+v+w=1$ 是在 uvw 空间上的一个顶点为（1，0，0），（0，1，0）和（0，0，1）的等边三角形。图 9.25 显示了一个三次三角形贝塞尔曲面，以及其控制点和控制多边形。



图 9.25　一个三次三角形B样条曲面

9.7.2 B 样条曲面

我们仅考虑一种类型的B样条曲面，即一种B样条矩形补面。B样条三角形补面的概念并不存在（Dahmen，Micchelli 和 Seidel 1992），而且也不在本书讨论的范围之内。

设$\{s_i\}_{i=0}^{n_0}$为单调集，即对所有i都有$s_i \leqslant s_{i+1}$。这些元素叫做结点，向量$(s_0, \cdots, s_{n_0})$叫做结点向量。类似地，设$\{t_i\}_{i=0}^{n_1}$为另一个单调集。给定一个三维空间控制点的矩形格子P_{i_0,i_1}（$0 \leqslant i_0 \leqslant n_0$，$0 \leqslant i_1 \leqslant n_1$），对这些点的B样条矩形补面为

$$X(s,t) = \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} B_{i_0,j_0}^{(0)}(s) B_{i_1,j_1}^{(1)}(t)\, P_{i_0,i_1}$$

其中$s \in [s_0, s_{n_0}]$，$t \in [t_0, t_{n_1}]$，$1 \leqslant j_0 \leqslant n_0$，$1 \leqslant j_1 \leqslant n_1$，并且表达式中的多项式满足 Cox-de Boor 公式：

$$B_{i,0}^{(0)}(s) = \begin{cases} 1, & s_i \leqslant s < s_{i+1} \\ 0, & 否则 \end{cases}$$

$$B_{i,j}^{(0)}(s) = \frac{(s - s_i) B_{i,j-1}^{(0)}(s)}{s_{i+j-1} - s_i} + \frac{(s_{i+j} - s) B_{i+1,j-1}^{(0)}(s)}{s_{i+j} - s_{i+1}}$$

且

$$B_{i,0}^{(1)}(t) = \begin{cases} 1, & t_i \leqslant t < t_{i+1} \\ 0, & 否则 \end{cases}$$

$$B_{i,j}^{(1)}(t) = \frac{(t - t_i) B_{i,j-1}^{(1)}(t)}{t_{i+j-1} - t_i} + \frac{(t_{i+j} - t) B_{i+1,j-1}^{(1)}(t)}{t_{i+j} - t_{i+1}}$$

$X(s,t)$的多项式分量实际上是在区间$[s_i, s_{i+1}] \times [t_j, t_{j+1}]$上分段定义的。多项式在每一个区间上的次数都为$i + j$。结点的值并不要求是均匀分布的。这种情形下的曲线叫做非均匀B样条曲面。如果结点的值是均匀分布的，曲线就叫做均匀 B 样条曲面。图 9.26 显示了一个均匀的双三次B样条曲面，以及其控制点和控制多边形。



图 9.26　一个均匀的双三次B样条曲面

9.7.3　非均匀有理 B 样条曲面

非均匀有理B 样条曲面（或称 NURBS 曲面）是从四维空间中的非均匀B样条多项式曲面中得出的。控制点为$(P_{i_0,i_1}, 1) \in \mathbb{R}^4$（$0 \leqslant i_0 \leqslant n_0$，$0 \leqslant i_1 \leqslant n_1$），多项式曲面为

$$(Y(s,t), w(s,t)) = \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} B_{i_0,j_0}^{(0)}(s) B_{i_1,j_1}^{(1)}(t)\, w_{i_0,i_1}(P_{i_0,i_1}, 1)$$

其中权重$w_{i_0,i_1} > 0$，$B_{i,j}^{(0)}(s)$和$B_{i,j}^{(1)}(s)$就是上一节中定义的同名多项式。NURBS 曲面可通过如下方法获得，即将（$Y(s,t), w(s,t)$）作为齐次向量，并除以最后一个分量，以得到在三维空间上的一个投影

$$X(s,t) = \frac{Y(s,t)}{w(s,t)} = \sum_{i_0=0}^{n_0} \sum_{i_1=0}^{n_1} R_{i_0,i_1,j_0,j_1}(s,t) P_{i_0,i_1}$$

其中

$$R_{i_0,i_1,j_0,j_1}(s,t) = \frac{w_{i_0,i_1} B_{i_0,j_0}^{(0)}(s) B_{i_1,j_1}^{(1)}(t)}{\sum_{k_0=0}^{n_0} \sum_{k_1=0}^{n_1} w_{k_0,k_1} B_{k_0,j_0}^{(0)}(s) B_{k_1,j_1}^{(1)}(t)}$$

第 10 章 三维距离

10.1 导言

假定有两个几何对象 $\mathcal{A}$ 和 $\mathcal{B}$，我们希望计算它们之间的距离。如果我们认为每一个对象都用参数函数 $A(\vec{s})$ 和 $B(\vec{t})$（$\vec{s} \in S \subset \mathbb{R}^m$ 且 $\vec{t} \in \mathcal{T} \subset \mathbb{R}^n$）来表示，那么存在一种通用的计算距离的方法。该方法就是，寻找在两个对象中所有可能的点对中，具有最小的距离平方的 $\mathcal{A}$ 上的一个点和 $\mathcal{B}$ 上的一个点。用函数来表示就是，$Q(\vec{s},\vec{t}) = \|A(\vec{s}) - B(\vec{t})\|^2$，其中 $(\vec{s},\vec{t}) \in S \times \mathcal{T} \subset \mathbb{R}^m \times \mathbb{R}^n$。问题的解就是该函数的最小值，解或者出现在 $S \times \mathcal{T}$ 的一个内点上，此时 $\nabla(Q) = \vec{0}$，或者出现在 $S \times \mathcal{T}$ 的一个边界上，此时该解使得一个二次函数具有最小值。除了介绍通用方法，我们还提供了针对一些特殊情况的求解方法，并且还分析了与问题相关的几何图元的几何性质。

10.2 点到线形对象的距离

假定有一个点 Q 和一条直线 $\mathcal{L}(t) = P + t\vec{d}$，我们要求 Q 与 $\mathcal{L}$ 之间的最小距离，如图 10.1 所示。如果我们观察 $\mathcal{L}$ 上与 Q 最近的点，并连接这两个点的线段，我们将发现，线段垂直于直线 $\mathcal{L}$。这一观察结果暗示我们可以使用点积，事实也确实如此，即最近的点 Q' 就是 Q 在直线 $\mathcal{L}$ 上的投影（如图 10.2 所示）。Q' 的参数值为

$$t_0 = \frac{\vec{d} \cdot (Q - P)}{\vec{d} \cdot \vec{d}} \tag{10.1}$$

而且

$$Q' = P + t_0\vec{d}$$

那么，从 Q 到 $\mathcal{L}$ 的距离为

$$\begin{aligned} d &= \|Q - Q'\| \\ &= \|Q - (P + t_0\vec{d})\| \end{aligned}$$

注意，如果 $\mathcal{L}$ 的方向 $\hat{d}$ 是规整的，我们有 $\|\hat{d}\| = 1$，因此方程（10.1）变成

$$t_0 = \hat{d} \cdot (Q - P)$$

并且还可以避免除法运算。

伪码为

```
float PointLineDistanceSquared3D(Point q, Line l, bool normalized, float& t)
{
```

```
    float distanceSquared;

    t = Dot(l.direction, VectorSubtract(q, l.direction));
    if (!normalized) {
        t /= Dot(l.direction, l.direction);
    }
    Point3D qPrime;

    qPrime = l.origin + t * l.direction;
    Vector3D vec = Q - qPrime;
    distanceSquared = Dot(vec, vec);

    return distanceSquared;
}
```

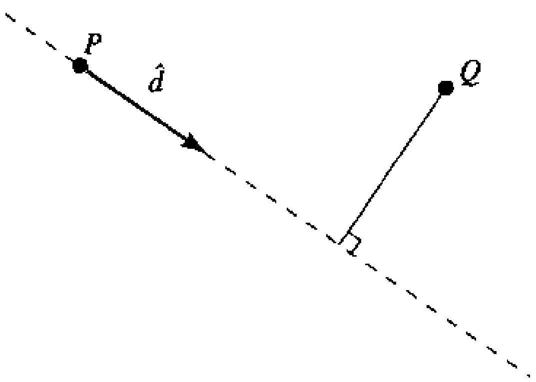


图 10.1　直线与点之间的距离

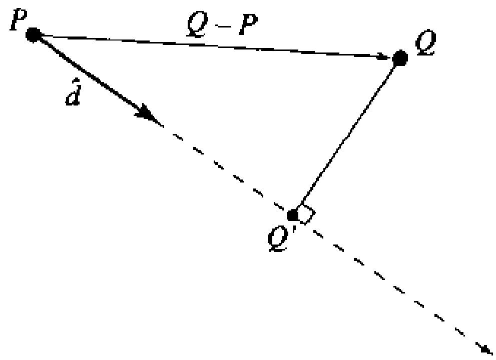


图 10.2　Q在$\mathcal{L}$上的投影

10.2.1　点到直线或射线的距离

如果$\mathcal{L}$是直线，那么我们刚才已经介绍了相关的求解方法。如果$\mathcal{L}$是射线，我们限制解只能为非负的值t_0；如果$t_0 < 0$，那么从Q到$\mathcal{L}$的距离为$\|Q-P\|$：

$$d = \begin{cases} \|Q-P\| & t_0 \leqslant 0 \\ \|Q-(Q+t_0\vec{d})\| & t_0 > 0 \end{cases}$$

如果$\mathcal{L}$是由两个端点P_0和P_1定义的线段，那么其方向向量可定义为

$$\vec{d} = P_1 - P_0$$

注意，这将使得$P_0 = \mathcal{L}(0)$，$P_1 = \mathcal{L}(1)$，因此，我们有

$$d = \begin{cases} \|Q-P_0\| & t_0 \leqslant 0 \\ \|Q-(P_0+t_0\vec{d})\| & 0 < t_0 < 1 \\ \|Q-(P_0+\vec{d})\| & t_0 \geqslant 1 \end{cases}$$

（如图 10.3 所示）。

用于射线情形的伪码为

```
float PointRayDistanceSquared3D(Point q, ray r, bool normalized, float& t)
{
    float distanceSquared;
```

```
    // Get distance to line - may have t < 0
    distanceSquared = PointLineDistanceSquared3D(q, r, normalized, &t);

    if (t < 0) {
        t = 0;
        // Get distance to ray origin instead
        Vector3D vec = q - r.origin;
        distanceSquared = Dot(vec, vec);
    }

    return distanceSquared;
}
```

用于线段情形的伪码为

```
float PointLineSegDistanceSquared3D(Point3D q, Segment3D s, bool normalized,
                                    float& t)
{
    float distanceSquared;

    // Get distance to line - may have t < 0 or t > 1
    distanceSquared = PointLineDistanceSquared3D(q, s, normalized, &t);
    if (t < 0) {
        t = 0;
        // Get distance to segment origin instead
        Vector3D vec = q - s.p0;
        distanceSquared = Dot(vec, vec);
    } else if (t > 1) {
        t = 1;
        // Get distance to segment terminus instead
        Vector3D vec = q - s.p1;
        distanceSquared = Dot(vec, vec);
    }

    return distanceSquared;
}
```

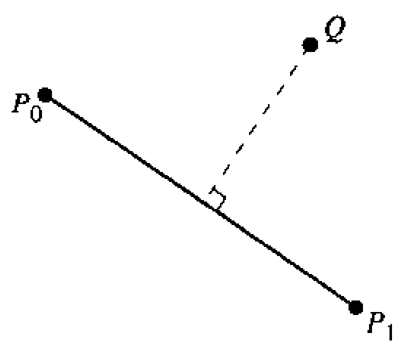


图 10.3　线段与点之间的距离

10.2.2　点到折线的距离

计算点 P 和顶点为 V_0 到 V_n，线段为 S_i（其端点为 V_i 到 V_{i+1}，$0 \leqslant i < n-1$）的折线 $\mathcal{L}$ 之间的距离的最直接的算法，是计算点与折线的所有线段之间距离的最小值。

$$\text{Distance}^2(P,\mathcal{L}) = \min_{0 \leqslant i < n-1} \text{Distance}^2(P,S_i) \tag{10.2}$$

对于具有大量线段的折线，或者涉及大量的折线并且需要经常计算距离的应用程序来说，盲目地迭代线段是非常费时的。

其实我们可以利用 6.2 节介绍的计算二维空间中点与折线之间距离的技术的三维扩展。这种方法迭代折线的线段并进行相对省时的线段排除，即排除那些不可能比当前最近的线段更近的线段。如果当前最近的线段为 S_c，它与点 P 之间的距离为 d。我们可以考虑一个球心位于 $P=(a,b,c)$，半径为 d 的球。任何与球不相交的线段与 P 之间的距离都不可能比 d 更近。然而，正如在 6.2 节中指出的那样，距离计算将用到我们希望避免的那种运算。在二维空间中，替代的方法是考虑包围圆的无穷带（参见图 6.4），并排除两个端点都位于同一个带内的折线线段。这种方法的三维空间模拟就是考虑包围球的厚板，如图 10.4 所示。

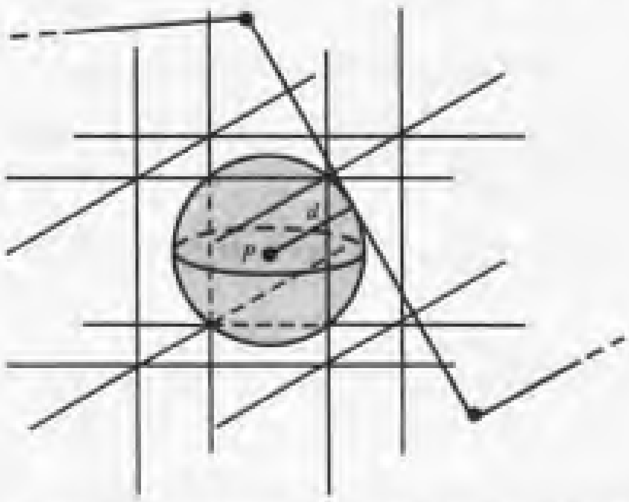


图 10.4　使用半空间法加快点与折线之间的距离测试

设 $S_i = ((x_i,y_i,z_i),(x_{i+1},y_{i+1},z_{i+1}))$ 为下一条将被测试的线段。如果 S_i 在无穷厚板 $|x-a| \leqslant d$ 之外，那么它不可能与圆相交。因此排除测试为

$$|x_i - a| \geqslant d \text{ 且 } |x_{i+1} - a| \geqslant d \text{ 且 } (x_i - a)(x_{i+1} - a) > 0$$

前面的两个条件保证线段的每一个端点都在厚板之外。最后一个条件保证端点都在厚板的同一侧。类似地，如果 S_i 在无穷厚板 $|y-b| \leqslant d$ 之外，那么它不可能与圆相交。因此排除测试为

$$|y_i - b| \geqslant d \text{ 且 } |y_{i+1} - b| \geqslant d \text{ 且 } (y_i - b)(y_{i+1} - b) > 0$$

最后，如果 S_i 在无穷厚板 $|z-c| \leqslant d$ 之外，那么它不可能与圆相交。因此排除测试为

$$|z_i - c| \geqslant d \text{ 且 } |z_{i+1} - c| \geqslant d \text{ 且 } (z_i - c)(z_{i+1} - c) > 0$$

图 10.5 说明了这一点。线段 S_0 产生折线与点 P 之间的当前最小距离 d 。线段 S_1 虽然位于圆外，但并未被排除，因为它部分地位于每一个厚板内。然而，S_2 被排除，因为它位于垂直厚板之外（为了清晰地说明，z 面并未画出）。

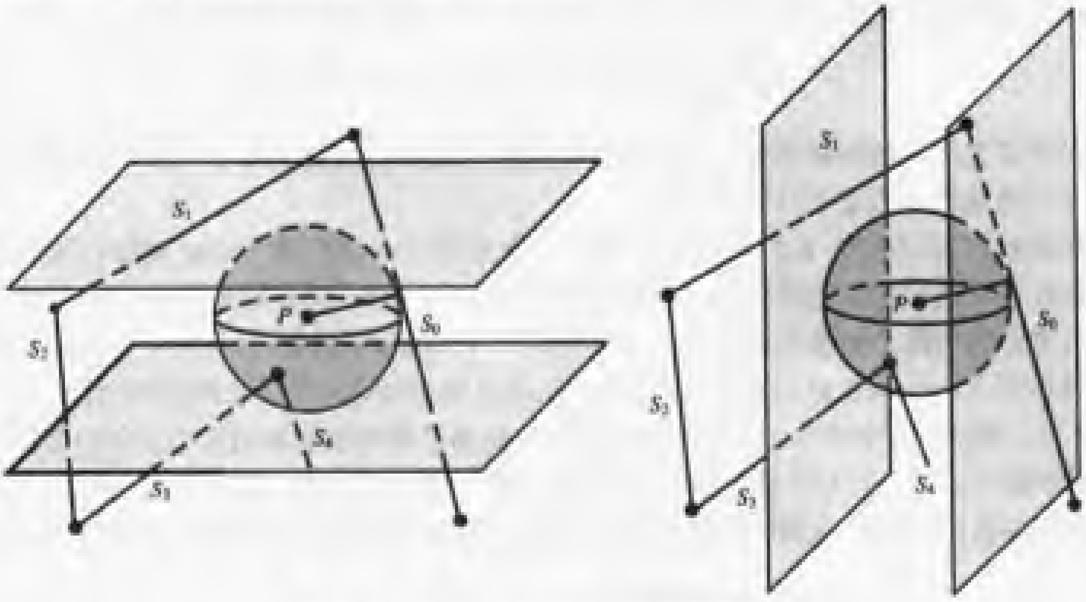


图 10.5　例子：点与折线之间距离的排除

由于在中间计算中应该避免平方根运算，因此一种实现方法保留 d^2 而不是 d 。排除测试也必须相应地使用 d^2 来重新建立

$$|x_i - a|^2 \geqslant d^2 \text{ 且 } |x_{i+1} - a|^2 \geqslant d^2 \text{ 且 } (x_i - a)(x_{i+1} - a) > 0$$

或

$$|y_i - b|^2 \geqslant d^2 \text{ 且 } |y_{i+1} - b|^2 \geqslant d^2 \text{ 且 } (y_i - b)(y_{i+1} - b) > 0$$

或

$$|z_i - c|^2 \geqslant d^2 \text{ 且 } |z_{i+1} - c|^2 \geqslant d^2 \text{ 且 } (z_i - c)(z_{i+1} - c) > 0$$

用于排除测试中的数量还可用于平方距离的计算，因此它们可以临时保存，以便于后来使用，这样可以避免重复计算。而且，当前排除测试中的数量 $x_{i+1} - a$，$y_{i+1} - b$ 和 $z_{i+1} - c$ 成为下一次排除测试中的 $x_i - a$，$y_i - b$ 和 $z_i - c$ 。因此这些数值应该临时保存，并在以后需要时使用，这也是为了避免重复计算。

伪码为

```
float PointPolylineDistanceSquared3D(Point p, Point vertices[],
                                     int nSegments)
{
    float dSq = INFINITY;
    float xMinusA, yMinusB, zMinusC;
    float xNextMinusA, yNextMinusB, zNextMinusC;
    float xMinusASq, yMinusBSq, zMinusCSq;
    float xNextMinusASq, yNextMinusBSq, zNextMinusCSq;
```

```
xMinusA = vertices[0].x - p.x;
yMinusB = vertices[0].y - p.y;
zMinusC = vertices[0].z - p.z;

xMinusASq = xMinusA * xMinusA;
yMinusBSq = yMinusB * yMinusB;
zMinusCSq = zMinusC * zMinusC;

xNextMinusA = vertices[1].x - p.x;
yNextMinusB = vertices[1].y - p.y;
zNextMinusC = vertices[1].z - p.z;

xNextMinusASq = xNextMinusA * xMNextinusA;
yNextMinusBSq = yNextMinusB * yNextMinusB;
zNextMinusCSq = zNextMinusC * zNextMinusC;

// Compute distance to first segment
Line l = { vertices[i], vertices[i+1] - vertices[i] };
float t;
dSq = PointLineDistanceSquared3D(p, l, FALSE, t)

// If closest point not on segment, check appropriate end point
if (t < 0) {
    dSq = MIN(dsq, xMinusASq + yMinusBSq + zMinusCSq);
} else if (t > 1) {
    dSq = MIN(dsq, xNextMinusASq + yNextMinusBSq + zNextMinusCSq);
}

// Go through each successive segment, rejecting if possible,
// and computing the distance squared if not rejected.
for (i = 1; i < nSegments - 1; i++) {
    // Rejection test
    if (((Abs(xMinusASq) > dSq) && (Abs(xNextMinusASq) <= dSq)
                                && (xMinusA * xNextMinusA > 0)) ||
        ((Abs(yMinusBSq) > dSq) && (Abs(yNextMinusBSq) <= dSq)
                                && (yMinusB * yNextMinusB > 0)) ||
        ((Abs(zMinusCSq) > dSq) && (Abs(zNextMinusCSq) <= dSq)
                                && (zMinusC * zNextMinusC > 0))) {

        if (i != nSegments - 2) {
            xMinusA = xNextMinusA;
            yMinusB = yNextMinusB;
            zMinusC = zNextMinusC;

            xNextMinusA = vertices[i + 2].x - p.x;
            yNextMinusB = vertices[i + 2].y - p.y;
            zNextMinusC = vertices[i + 2].z - p.z;
        }

        continue;
    }
    // Rejection test failed - check distance to line
```

```
            Line  l = { vertices[i], vertices[i+1] - vertices[i] };
            float t;
            dSq = PointLineDistanceSquared3D(p, l, FALSE, t)

            // If closest point not on segment, check appropriate end point
            if (t < 0) {
               dSq = MIN(dsq, xMinusASq + yMinusBSq + zMinusCSq);
            } else if (t > 1) {
               dSq = MIN(dsq, xNextMinusASq + yNextMinusBSq + zNextMinusCSq);
            }

            if (i != nSegments - 2) {
                xMinusA = xNextMinusA;
                yMinusB = yNextMinusB;
                zMinusC = zNextMinusC;

                xNextMinusA = vertices[i + 2].x - p.x;
                yNextMinusB = vertices[i + 2].y - p.y;
                zNextMinusC = vertices[i + 2].z - p.z;
            }
        }

        return dSq;
    }
```

排除测试的一种改进是利用线段与包含圆心为 $\boldsymbol{P}$、半径为 d 的圆的轴平行框的相交测试。我们可以使用 11.11 节讨论的轴分解方法。图 10.5 说明了这种改进的方法。前一种方法没有排除线段 S_1，因为它部分地位于每一个厚板内。然而，用当前的方法可排除 S_1，因为它与轴平行矩形不相交。

10.3 点到平面对象的距离

在本节中，我们将讨论点与平面对象（平面、三角形、矩形、多边形、圆和圆盘）之间的距离的计算问题。

10.3.1 点到平面的距离

在本节中，我们考虑点 Q 到平面 $\mathcal{P}$：$\{P,\vec{n}\}$ 的距离，其中 $\vec{n}$ 为平面的法线向量，且 P 为平面 $\mathcal{P}$ 上的一个点（9.2.1 节），如图 10.6 所示。平面 $\mathcal{P}$ 上最接近 Q 的点记为 Q'。注意，Q 与 Q' 构成的向量垂直于平面 $\mathcal{P}$（即平行于 $\vec{n}$），并且我们可根据这一事实来研究 Q 与 $\mathcal{P}$ 之间的距离。

图 10.7 显示了平面 $\mathcal{P}$ 的一个横截面。根据三角几何，我们可得到

$$\cos\theta = \frac{\|Q - Q'\|}{\|Q - P\|}$$

因此，我们有

$$\begin{aligned} d &= \|Q - Q'\| \\ &= \|Q - P\|\cos\theta \end{aligned} \tag{10.3}$$

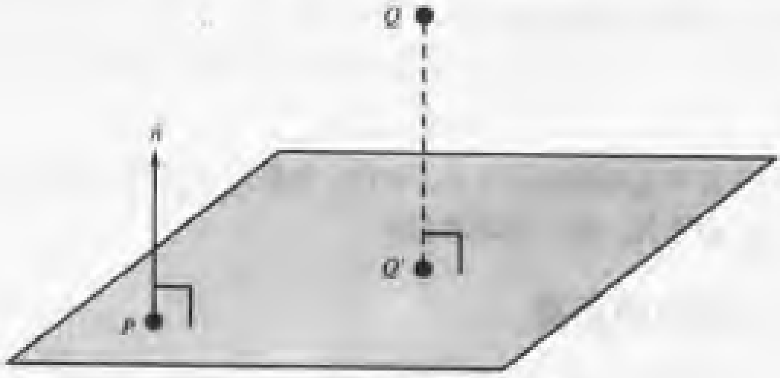


图 10.6 点与平面之间的距离

根据定义，$\vec{u}\cdot\vec{v}=\|\vec{u}\|\|\vec{v}\|\cos\theta$，因此，我们可以将方程（10.3）改写为

$$\begin{aligned}d&=\|Q-P\|\cos\theta\\&=\|Q-P\|\frac{-k\vec{n}\cdot-(Q-P)}{\|k\vec{n}\|\|Q-P\|}\\&=\frac{|\vec{n}\cdot(Q-P)|}{\|\vec{n}\|}\end{aligned}$$

如果平面的法线是单位长度的，那么分母为 1，不需要进行除法运算。

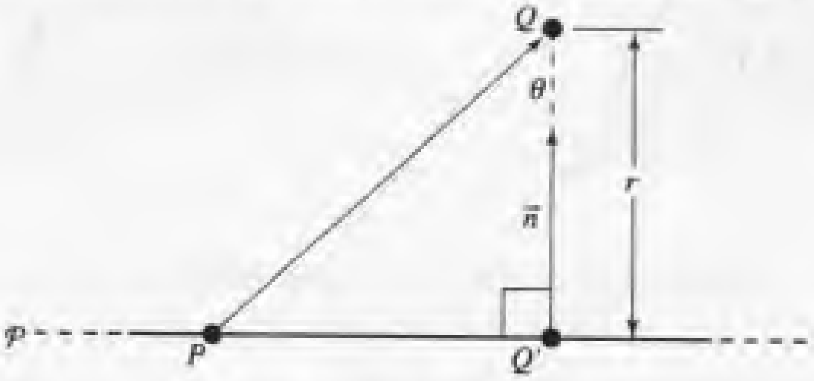


图 10.7 平面$\mathcal{P}$的边缘视图

在有的情形中，点位于平面的哪一面是重要的。此时，可能需要有符号的距离（符号与平面的法线相关）。如果平面方程是规则的，那么有一个简单的算法可以得到有符号的距离（Georgiades 1992）。如果我们有一个用“向量版本”的隐含形式所表示的平面$\mathcal{P}$

$$P\cdot\hat{n}+d=0$$

并且点 Q 与其在平面$\mathcal{P}$上的投影Q'之间的连线与 $\hat{n}$ 平行。如果平面法线是规则的，那么$\|Q-Q'\|$为 $\hat{n}$的数值倍数，即数值倍数就是距离。

根据 Georgiades（1992），推导如下：设$r=\|Q-Q'\|$，则有

$$Q=r\hat{n}+Q'$$

如果我们用 $\hat{n}$ 乘以方程的两边，可得

$$\hat{n}\cdot Q=r\hat{n}\cdot\hat{n}+\hat{n}\cdot Q' \tag{10.4}$$

然而，由于我们已假设法线是规整的，因此 $\|\hat{n}\| = 1$。而且，由于 Q' 定义为在平面上，因此根据定义有 $\hat{n} \cdot Q' = -d$。如果我们将上述的两式代入方程（10.4），可得

$$r = \hat{n} \cdot Q + d$$

因此，点 Q 与平面 $\mathcal{P}$ 之间的距离为 r，而且，如果点 Q 在平面法线所指向的一面，那么距离为正，如果点 Q 在另一面，则距离为负。

10.3.2 点到三角形的距离

在本节中，我们讨论点 P 与三角形之间的距离的计算问题，如图 10.8 所示。为了本节讨论的方便，我们用参数形式来定义顶点为 $\{V_0, V_1, V_2\}$ 的三角形 $\mathcal{T}$：

$$\mathcal{T}(s,t) = B + s\vec{e}_0 + t\vec{e}_1$$

其中，$(s,t) \in D = \{(s,t) : s \in [0,1], t \in [0,1], s + t \leqslant 1\}$，$B = V_0$，$\vec{e}_0 = V_1 - V_0$ 且 $\vec{e}_1 = V_2 - V_0$。通过确定对应于三角形上最接近于点 P 的点 P' 的值 $(\bar{s}, \bar{t}) \in D$，可以计算出最小距离。

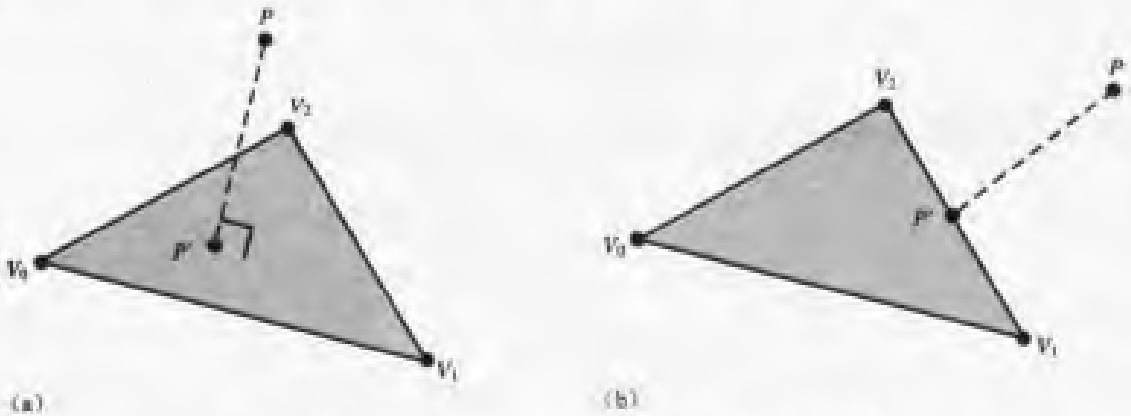


图 10.8　点与三角形之间的距离。最近的点可能在三角形内
（a）在三角形的边上（b）或是三角形的一个顶点

三角形上任意的一个点与点 P 之间的距离为

$$\|\mathcal{T}(s,t) - P\|$$

但是我们使用距离平方函数

$$Q(s,t) = \|\mathcal{T}(s,t) - P\|^2$$

其中 $(s,t) \in D$。如果我们扩展各项，并将它们乘出来，可以看出，该函数是关于 s 和 t 的二次函数

$$Q(s,t) = as^2 + 2bst + ct^2 + 2ds + 2et + f$$

其中

$$a = \vec{e}_0 \cdot \vec{e}_0$$

$$b = \vec{e}_0 \cdot \vec{e}_1$$

$$c = \vec{e}_1 \cdot \vec{e}_1$$

$$d = \vec{e}_0 \cdot (B - P)$$
$$e = -\vec{e}_1 \cdot (B - P)$$
$$f = (B - P) \cdot (B - P)$$

二次方程可根据$ac - b^2$来分类。对于 Q，有

$$\begin{aligned} ac - b^2 &= (\vec{e}_0 \cdot \vec{e}_0)(\vec{e}_1 \cdot \vec{e}_1) - (\vec{e}_0 \cdot \vec{e}_1)^2 \\ &= \|\vec{e}_0 \times \vec{e}_1\|^2 \\ &> 0 \end{aligned}$$

由于我们假设三角形的两条边$\vec{e}_0$和$\vec{d}_1$是线形无关的（即不平行且长度都不为零），因此该值为正。所以，它们的叉积为一个非零的向量。

用微积分的术语来说，我们的目标就是在定义域D上求函数$Q(s,t)$的最小值。由于Q是一个连续且可微的函数，其最小值或者出现在D的一个梯度为$\nabla Q = 2(as + bt + d, bs + ct + e) = (0,0)$的内点，或者出现在$D$的边界上。

只有当下式成立时，Q的梯度才为零

$$\bar{s} = \frac{be - cd}{ac - b^2}$$

并且

$$\bar{t} = \frac{bd - ad}{ac - b^2}$$

如果$(\bar{s},\bar{t}) \in D$，那么我们已得到Q的最小值。否则，最小值必定出现在三角形的边界上。为了找到正确的边界，考虑图 10.9。位于中心的标号为 0 的三角形就是Q的定义域，$(s,t) \in D$。如果$(\bar{s},\bar{t})$在区域 0 内，那么三角形上最接近于P的点位于三角形内。

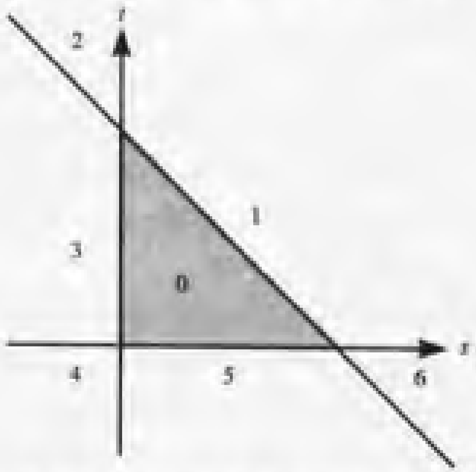


图 10.9 用三角形区域对平面st分区

假设$(\bar{s},\bar{t})$位于区域 1 内。Q的阶层曲线就是st平面上Q为常数的曲线。由于Q的图形为抛物面，因此阶层曲线为椭圆（参见 A.9.1 节）。在$\nabla Q = (0,0)$的点上，阶层曲线退化为一个单点$(\bar{s},\bar{t})$。Q的全局最小值就出现在该点上，设该最小值为$V_{\min}$。由于阶层值V从$V_{\min}$开始增加，因此对应的椭圆从$(\bar{s},\bar{t})$向外增长。最小的阶层值V_0所对应的椭圆（由$Q = V_0$隐

式定义）刚好与三角形定义域的边$s+t=1$接触于$s=s_0\in[0,1]$，$t_0=1-s_0$。对于$V<V_0$的阶层值，对应的椭圆与D不相交。对于$V>V_0$的阶层值，D的一部分位于对应的椭圆内。特别地，这些椭圆与边的任意交点必定具有阶层值$V>V_0$。因此，$Q(s,1-s)>Q(s_0,t_0)$（$s\in[0,1]$且$s\neq s_0$）。点(s_0,t_0)提供了P与平面之间的距离平方的最小值。三角形点是一个边界点。图 10.10 通过显示不同的阶层曲线来说明了这一点。

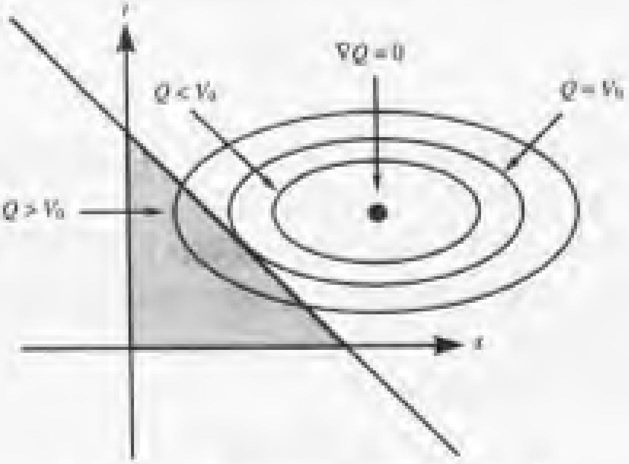


图 10.10　几种不同的阶层曲线$Q(s,t)=V$

直观表示出现在边界上的最小距离点的另一种方法是求位于(s,t,Q)空间上平面$s+t=1$与Q的图形的交点。相交的曲线是一条抛物线，就是$F(s)=Q(s,1-s)$（$s\in[0,1]$）的图形。现在，问题简化为在一维上求函数$F(s)$（$s\in[0,1]$）的最小值。F的最小值或者出现在$[0,1]$的一个内点，此时在该点上有$F'(s)=0$，或者出现在一个端点$s=0$或$s=1$上。图 10.10 显示了最小值出现在内点上的情形。椭圆与直线$s+t=1$相切于该点。对于端点的情形，椭圆可能刚好与D的一个顶点相触，但不一定相切。

为了区分内点和端点的不同情形，可以对一维的情形使用相同的分区方法。区间$[0,1]$将实线划分为三个区间：$s<0, s\in[0,1]$，以及$s>1$。设$F'(\hat{s})=0$。如果$\hat{s}<0$，那么$F(s)$在$s\in[0,1]$上是递增函数。它在$[0,1]$内的最小值必定出现在$s=0$处，此时Q在$(s,t)=(0,1)$处得到其最小值。如果$\hat{s}>1$，那么$F(s)$在$s\in[0,1]$上是递减函数。它在$[0,1]$内的最小值必定出现在$s=1$处，此时Q在$(s,t)=(1,0)$处得到其最小值。否则，$\hat{s}\in[0,1]$，F在$\hat{s}$处得到最小值，而Q在$(s,t)=(\hat{s},1-\hat{s})$处得到其最小值。

$(\bar{s},\bar{t})$出现在区域 3 或区域 5 中的情况的处理方法与全局最小值在区域 0 内的处理方法相同。如果$(\bar{s},\bar{t})$在区域 3 内，那么最小值出现在$(s_0,0)$（$s_0\in[0,1]$）处。确定第一个接触点是对应区间的内点还是端点的方法与前面讨论的方法相同。

如果$(\bar{s},\bar{t})$在区域 2 内，提供与单位正方形的第一个接触点的Q的阶层曲线将或者与边$s+t=1$或者与边$s=0$相接触。由于全局最小值出现在区域 2 中，并且Q的阶层曲线集都是椭圆，因此，至少方向导数$(0,-1)\cdot\nabla Q(0,1)$和$(1,-1)\cdot\nabla Q(0,1)$中的一个必定是正的。两个向量$(0,-1)$和$(1,-1)$分别是边$s=0$和边$s+t=1$的方向向量。可基于$(0,-1)\cdot\nabla Q(0,1)$和$(1,-1)\cdot\nabla Q(0,1)$的符号来决定选择边$s+t=1$还是边$s=0$。

对区域 6 也可以进行相同的讨论。在区域 4 中，两个数$(1,0)\cdot\nabla Q(0,0)$和$(0,1)\cdot\nabla Q(0,0)$

的符号确定哪条边包含最小值。

算法的实现被设计为在计算最小距离和对应的最接近点时最多只需进行一次浮点除法运算。而且，除法被推迟到需要时才进行。在有的情形中，并不需要进行除法运算。

首先计算所有代码都需使用的数值。特别地，要计算的数值有

$$\vec{d} = B - P$$

$$a = \vec{e}_0 \cdot \vec{e}_0$$

$$b = \vec{e}_0 \cdot \vec{e}_1$$

$$c = \vec{e}_1 \cdot \vec{e}_1$$

$$d = \vec{e}_0 \cdot \vec{d}$$

$$e = \vec{e}_1 \cdot \vec{d}$$

$$f = \vec{d} \cdot \vec{d}$$

由于对于一些边长小的边来说，有些浮点舍入错误将导致一个小的负数，因此，代码计算的实际上是$\sigma = |ac - b^2|$。

在理论推导时，我们计算满足$\nabla Q(\bar{s}, \bar{t}) = (0, 0)$的$\bar{s} = (be - cd)/\sigma$和$\bar{t} = (bd - ae)/\sigma$。然后测试全局最小值的位置，以确定它是否是一个三角形定义域$\boldsymbol{D}$。如果是，那么，我们已经确定计算最小距离所需的数值。如果不是，那么，必须测试D的边界。为了推迟除以σ的除法，代码仅仅计算$\bar{s} = be - dc$和$\bar{t} = bd - ae$，并测试在一个放大的定义域（$s \in [0, \sigma]$，$t \in [0, \sigma]$且$s + t \leqslant \sigma$）内的包含性。如果在该集合内，则执行除法。如果不在，则测试单位正方形的边界。确定哪条边包含$(\bar{s}, \bar{t})$的条件的一般形式为

```
det = a*c - b*b;  s = b*e - c*d;  t = b*d - a*e;
if (s + t <= det) {
    if (s < 0) {
        if (t < 0) {
            region 4
        } else {
            region 3
        }
    }
} else {
    if (s < 0) {
        region 2
    } else if (t < 0) {
        region 6
    } else {
        region 1
    }
}
```

处理区域 0 的代码段列出如下

```
invDet = 1 / det;
s *= invDet;
```

```
t *= invDet;
```

并且需要进行一次除法。

处理区域 1 的代码列出如下。

```
// F(s) = Q(s, 1 - s) = (a - 2b + c)s^2 + 2(b - c + d - e)s + (c + 2e + f)
// F'(s)/2 = (a - 2b + c)s + (b - c + d - e)
// F'(s) = 0 when s = (c + e - b - d)/a - 2b + c)
// a - 2b + c = |e0 - e1|^2 > 0,
// so only the sign of c + e - b - d need be considered

numer = c + d - b - d;

if (numer <= 0) {
    s = 0;
} else {
    denom = a - 2 * b + c;  // positive quantity
    s = (numer >= denom ? 1: numer/denom);
}
t = 1 - s;
```

处理区域 3 的代码段列出如下。处理区域 5 的代码段与此类似。

```
// F(t) = Q(0, t) = ct^2 + et + f
// F'(t)/2 = ct + e
// F'(t) = 0 when t = -e/c

s = 0;
t = (e >= 0 ? 0 : (-e >= c ? 1: -e/c));
```

处理区域 2 的代码段列出如下。处理区域 4 和区域 6 的代码段与此类似。

```
// Grad(Q) = 2(as + bt + d, bs + ct + e)
// (0, -1) * Grad(Q(0, 1)) = (0, -1) * (b + d, c + e) = -(c + e)
// (1, -1) * Grad(Q(0, 1)) = (1, -1) * (b + d, c + e) = (b + d) - (c + e)
// min on edge s + t = 1 if (1, -1) * Grad(Q(0, 1)) < 0)
// min on edge s = 0 otherwise

tmp0 = b + d;
tmp1 = c + e;
if (tmp1 > tmp0) {  // min on edge s + t = 1
    numer = tmp1 - tmp0;
    denom = a - 2 * b + c;
    s = (numer >= denom? 1 : numer / denom);
    t = 1 - s;
} else {
    s = 0;
    t = (tmp1 <= 0 ? 1 : (e >= 0 ? 0 : -e / c));
}
```

10.3.3　点到矩形的距离

一般地，矩形由 4 个顶点 P_0，P_1，P_2和P_3所定义。然而，如果我们设 $P=P_0$，并定义 $\vec{e}_0=P_1-P$ 和 $\vec{e}_1=P_3-P$，那么，矩形可等价定义为 $\mathcal{R}(s,t)=P+s\vec{e}_0+t\vec{e}_1$（$(s,t)\in[0,1]^2$），如图 10.11 所示。



图 10.11　矩形的另一种定义

给定点 Q 和矩形 $\mathcal{R}$，我们希望找到 Q 与 $\mathcal{R}$ 上最接近 Q 的点 Q' 之间的距离，如图 10.12 所示。矩形上最接近于 Q 的点可以通过如下方法得到：将 Q 投影到包含 $\mathcal{R}$ 的平面上，然后分析 Q' 与 $\mathcal{R}$ 的顶点和边之间的关系。如果 Q' 是 $\mathcal{R}$ 的外点，那么，最接近的点不是 $\mathcal{R}$ 的一个顶点就是 $\mathcal{R}$ 的一条边上的点。向量 $\vec{e}_0$ 和 $\vec{e}_1$ 是正交的，并且如果扩展到区域 $[0,1]$ 之外，则它们可将平面划分为 9 个区域，如图 10.13 所示。

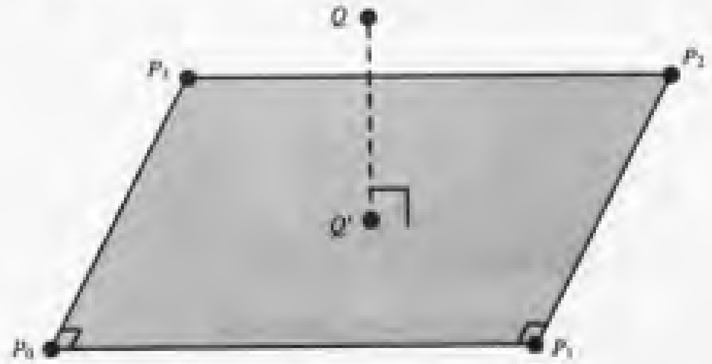


图 10.12　点与矩形之间的距离

如果投影点在区域 0 内，那么投影点就是最接近于 Q 的点。如果它在区域 2，4，6 或 8 内，那么，最接近的点分别为 P_2，P_3，P_0 或 P_1。最后，如果它在区域 1，3，5 或 7 内，那么，最接近的点就是该点分别在边 P_1P_2，P_2P_3，P_3P_0 或 P_0P_1 上的投影。

Q 在包含 $\mathcal{R}$ 的平面上的投影为

$$Q'=Q+s\vec{e}_0+t\vec{e}_1$$

其中

$$s=(Q-P)\cdot\vec{e}_0 \tag{10.5}$$

$$t = (Q - P) \cdot \vec{e}_1 \tag{10.6}$$



图 10.13　用矩形对平面分区

伪码为

```
float PointRectangleDistanceSquared3D(Point q, Rectangle rectangle)
{
    float d = q - rectangle.p;

    float s = Dot(rectangle.e0, d);
    if (s > 0) {
        float dot0 = Dot(rectangle.e0, rectangle.e0);
        if (s < dot0) {
            d = d - (s / dot0) * rectangle.e0;
        } else {
            d = d - rectangle.e0;
        }

    }

    float t = Dot (rectangle.e1, d);
    if (t > 0) {
        float dot1 = Dot(rectangle.e1, rectangle.e1);
        if (t < dot1) {
            d = d - (t / dot1) * rectangle.e1;
        } else {
            d = d - rectangle.e1;
        }
    }

    return Dot(d, d);
}
```

10.3.4　点到多边形的距离

在本节中，我们将讨论计算三维空间中点与多边形之间的距离问题，如图 10.14 所示。

多边形可以定义为 n 个顶点序列：V_0，V_1，…，V_{n-1}。这些点所在的平面可以由如下方法来计算：找出三个不共线的点，V_i，V_j，V_k，$i<j<k$，将叉积 $(V_j-V_i)\times(V_k-V_i)$ 作为平面的法线，将任意点 V_i（$0\leqslant i\leqslant n-1$）作为平面上的一个点。浮点数存在误差，然而确定不共线需要精确的计算，因此，最好使用 A.7.4 节中描述的 Newell 方法（Tampieri 1992）。

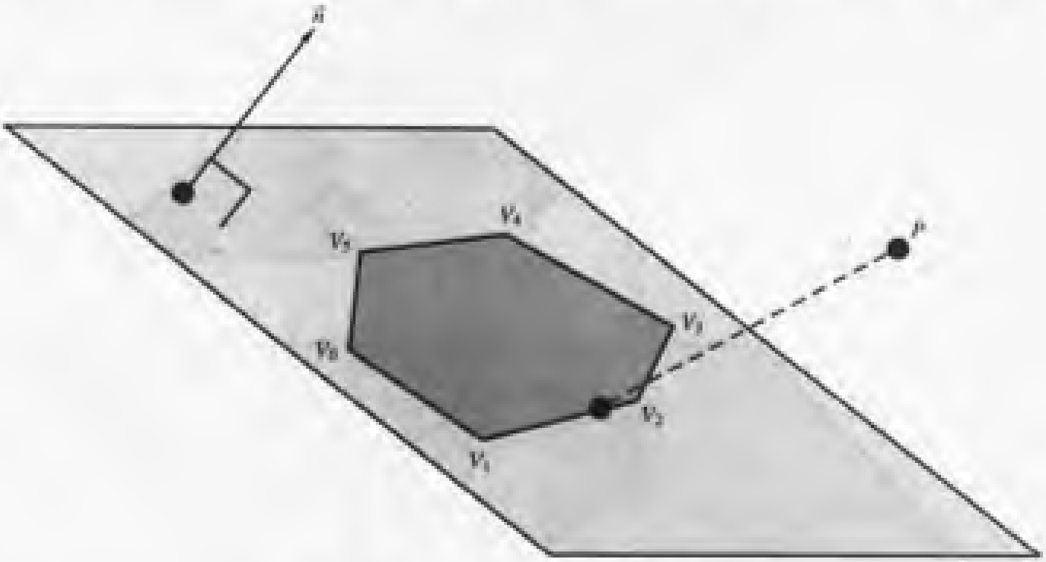


图 10.14 点到多边形的距离

设多边形 $\mathcal{P}$ 所在的平面定义为 $ax+by+cz+d=0$。点 P 在 $\mathcal{P}$ 上的投影为

$$P'=P-\frac{P\cdot\vec{n}+d}{\vec{n}\cdot\vec{n}}\vec{n}$$

那么，如果我们将 V_i 和 P' 投影到 XY，XZ 或 YZ 中的一个平面上（一般地，我们沿与多边形各点距离最小的轴进行投影），那么可以利用 6.3.4 节介绍的二维空间中点与多边形之间距离的算法，计算出投影所得到的多边形上最接近于 P'' 的点 Q'。未投影的多边形上最接近于 P 的点 Q 可通过将 Q' 的坐标代入多边形的平面方程中而计算出来。于是，P 与 Q 之间的距离就是从 P 到多边形的距离（如图 10.15 所示）。

伪码为

```
float PointPolygonDistanceSquared3D(Point p, Polygon poly)
{
    // Get plane equation for polygon
    float a, b, c, d;
    PolygonPlaneEquation(poly, a, b, c, d);
    Vector n = { a, b, c };

    // Project point onto plane of polygon
    Point pPrime;
    pPrime = p - ((Dot(p, n) - d) / Dot(n, n)) * n;

    // Determine plane to project polygon onto
    if (MAX(n.x, MAX(n.y, n.z)) == n.x) {
        projectionPlane = YZ_PLANE;
    } else if (MAX(n.x, MAX(n.y, n.z)) == n.y) {
```

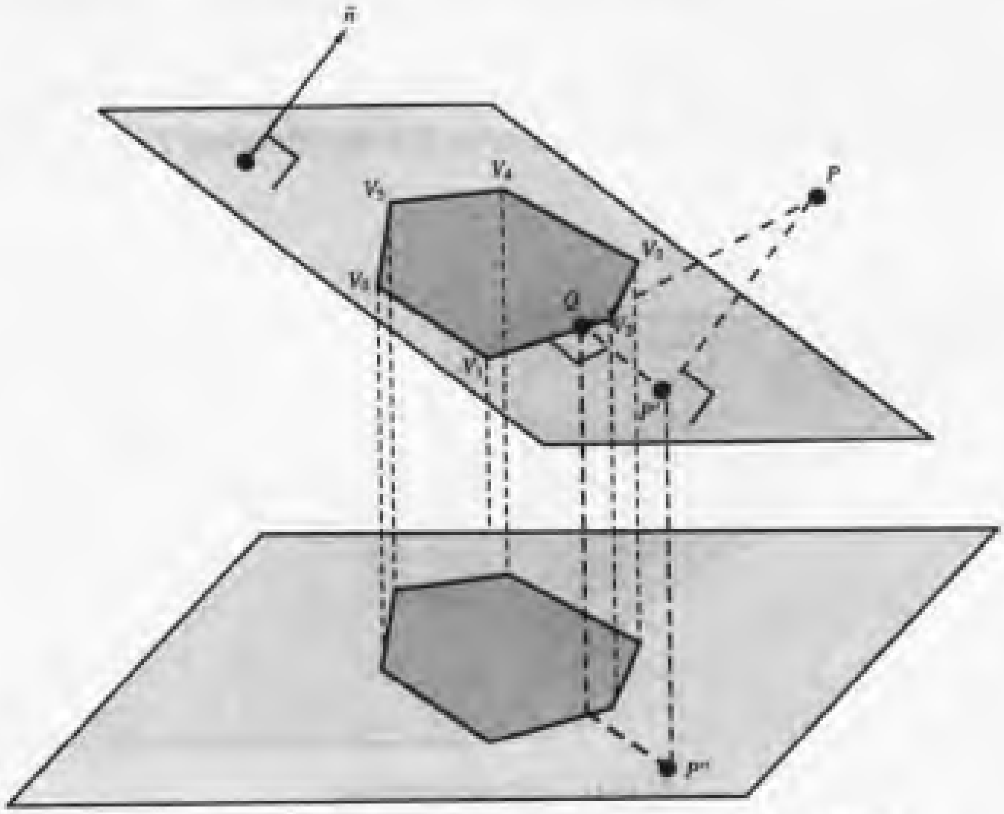


图 10.15　用投影到二维空间的方法求解三维空间点与多边形的距离

```
    projectionPlane = XZ_PLANE;
} else {
    projectionPlane = XY_PLANE;
}

// Project poly and pPrime onto plane
Point pPrimePrime = pPrime;
if (projectionPlane == YZ_PLANE) {
    pPrimePrime.x = 0;
} else if (projectionPlane == XZ_PLANE) {
    pPrimePrime.y = 0;
} else {
    pPrimePrime.z = 0;
}

Polygon2D poly2D;
for (i = 0; i < poly.nVertices; i++) {
    poly2D.vertices[i] = poly.vertices[i];
    if (projectionPlane == YZ_PLANE) {
        poly2D.vertices[i].x = 0;
    } else if (projectionPlane == XZ_PLANE) {
        poly2D.vertices[i].y = 0;
    } else {
        poly2D.vertices[i].z = 0;
```

```
        }
    }

    // Find closest point in 2D
    Point qPrime;
    float dist2D;
    dist2D = PointPolygonDistance2D(pPrimePrime, poly2D, qPrime);

    // Compute q, the closest point on the 3D polygon's plane
    q = qPrime;
    if (projectionPlane == YZ_PLANE) {
        qPrime.x = (-b * qPrime.y - c * qPrime.z - d) / a;
    } else if (projectionPlane == XZ_PLANE) {
        qPrime.y = (-a * qPrime.x - c * qPrime.z - d) / b;
    } else {
        qPrime.z = (-a * qPrime.x - b * qPrime.y - d) / c;
    }

    // Finally, compute distance (squared)
    Vector3D d = p - qPrime;
    return Dot(d, d);
}
```

10.3.5 点到圆或圆盘的距离

三维空间中的圆用圆心C，半径r，以及包含圆的平面$\hat{n}\cdot(X-C)=0$（其中$\hat{n}$为平面的单位长度法线）来表示。如果$\hat{u}$和$\hat{v}$也是单位长度向量，并且$\hat{u}$，$\hat{v}$和$\hat{n}$构成一个右手正交坐标系（这些向量作为列组成的矩阵是行列式为 1 的正交矩阵），那么圆的参数方程为

$$X = C + r(\cos(\theta)\hat{u} + \sin(\theta)\hat{v}) =: C + r\hat{w}(\theta)$$

其中角度$\theta\in[0,2\pi)$。注意$\|X-C\|=r$，所有的X与C的距离都相等。而且，$\hat{n}\cdot(X-C)=0$，由于$\hat{u}$和$\hat{v}$与$\hat{n}$都垂直，因此X在平面上。

对于每一个$\theta\in[0,2\pi)$，从指定点P到对应的圆上的点的距离平方为

$$F(\theta)=\|C+r\hat{w}(\theta)-P\|^2=r^2+\|C-P\|^2+2r(C-P)\cdot\hat{w}$$

需要求解的问题是如何通过找到θ_0使得$F(\theta_0)\leqslant F(\theta)$来求$F(\theta)$的极小值。由于$F$是可微的周期函数，因此其极小值必定出现在$F'(\theta)=0$处。同时，注意$(C-P)\cdot\hat{w}$应该为负，其数量应该尽可能地大，以减小$F$的定义式中右边的值。其导数为

$$F'(\theta)=2r(C-P)\cdot\hat{w}'(\theta)$$

由于对所有的θ，$\hat{w}\cdot\hat{w}=1$，所以其中的$\hat{w}\cdot\hat{w}'=0$。由于$\hat{w}''=-\hat{w}$和$0=\hat{w}\cdot\hat{w}'$隐含说明了$0=\hat{w}\cdot\hat{w}''+\hat{w}'\cdot\hat{w}'=-1+\hat{w}'\cdot\hat{w}'$，因此$\hat{w}'$是单位长度向量。最后，由于$\hat{n}\cdot\hat{w}=0$隐含说明了$0=\hat{n}\cdot\hat{w}'$，因此$\hat{w}'$垂直于$\hat{n}$。所有的条件隐含说明了$\hat{w}$平行于$P-C$在平面上的投影，且指向的方向相同。

设Q为P在平面上的投影，那么

$$Q - C = P - C - \left(\hat{n} \cdot (P - C)\right)\hat{n}$$

向量$\hat{w}(\theta)$必定为投影$(Q - C)/\|Q - C\|$的单位向量。圆上与P最接近的点就是

$$X = C + r\frac{Q - C}{\|Q - C\|}$$

其中假定$Q \neq C$。于是，点到圆的距离为$\|P - X\|$。

如果P的投影刚好就是圆心C，那么圆上的所有点都与C等距。点到圆的距离就是顶点为C、P与圆上的任意一点的直角三角形之斜边的长度。相邻和相对的三角形边的长度为r和$\|P - C\|$，因此点到圆的距离为$\sqrt{r^2 + \|P - C\|^2}$。

图 10.16 显示了P的投影不在圆心的典型情形。P投影到圆心的情形如图 10.17 所示。

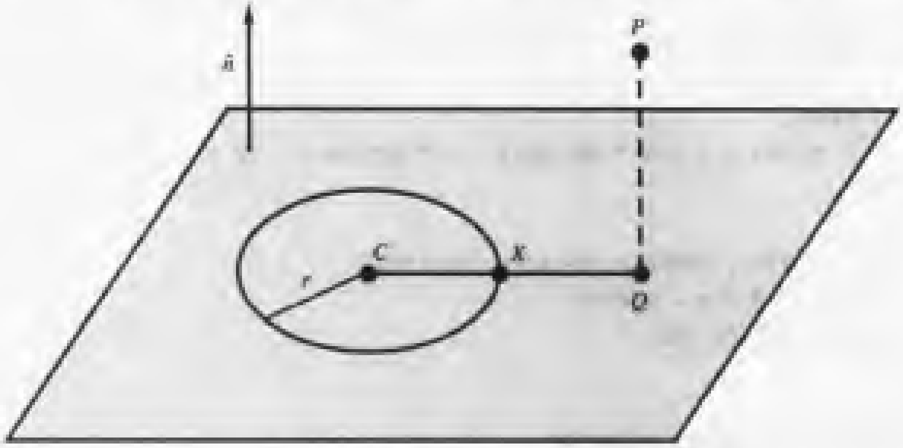


图 10.16　典型例子：距圆最近的点

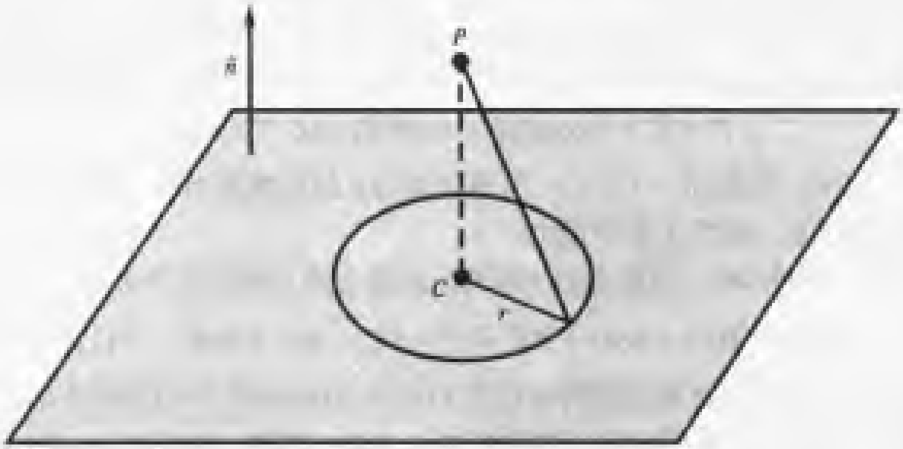


图 10.17　投影为圆心时

点到圆盘的距离

要计算点到圆盘的距离，就应该对计算点到圆的距离的算法做一点微小的修改。圆盘是满足$X = C + \rho\hat{w}(\theta)$（$0 \leqslant \rho \leqslant r$）的所有点的集合。如果$P$的投影在圆盘内，那么投影点就是最接近于$P$的点。如果投影在圆盘的外面，那么最接近于$P$的点就是圆盘的边界（即圆）上最接近于$P$的点。图 10.18 显示了$P$投影在圆盘内的情形，而图 10.19 显示了$P$投影在圆盘外的情形。

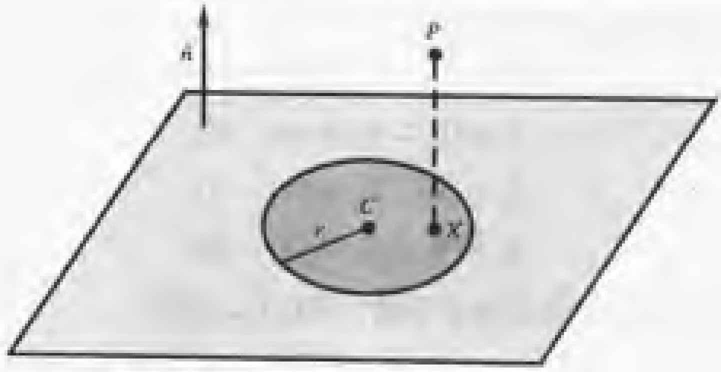


图 10.18 当 P 投影在圆盘内时的最近点

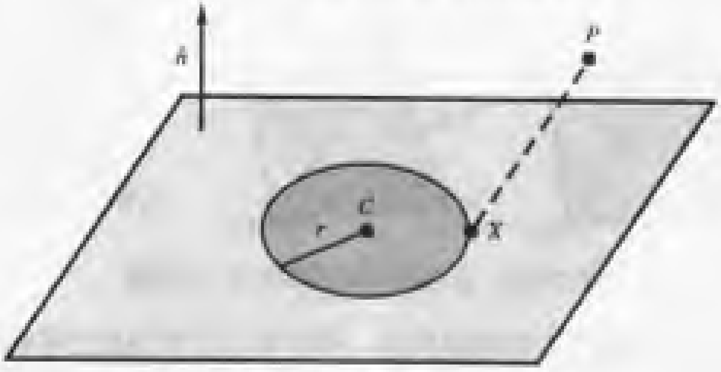


图 10.19 当 P 投影在圆盘外时的最近点

10.4 点到多面体的距离

在本节中，我们将讨论点到多面体的距离。我们首先考虑一般的情形，然后讨论两种常见的特殊情形，即点到有向有界箱（OBB）和正交平截体的距离。

10.4.1 一般问题

本节讨论点到多面体的距离。我们提供了三个算法，一个针对四面体、一个针对严格凸多面体，另一个针对凹多面体。

1. 点到四面体的距离

我们将四面体作为本节所讨论的问题的特殊情形，因为在计算它到点的距离时能揭示其性质。考虑一个点 P，以及一个具有不共面顶点 V_i（$0 \leqslant i \leqslant 3$）的四面体，如图 10.20 所示。该点可能最接近于一条边、一个面的一个内点，或者其中的一个顶点（或者它可能位于多面体的内部，此时距离为零）。为了简化讨论，我们假定顶点是有序的，且按照这种次序，各列分别为 $V_i - V_0$（$0 \leqslant i \leqslant 2$）的 3×3 的矩阵 $\mathbf{M}$ 具有正的行列式。具有此次序的规范四面体为

$$V_0 = (0, 0, 0)$$

$$V_1 = (1, 0, 0)$$

$$V_2 = (0, 1, 0)$$

$$V_3 = (0, 0, 1)$$

它的三角形面的指向外的法线向量为

$$\vec{n}_0 = (V_1 - V_3) \times (V_2 - V_3)$$

$$\vec{n}_1 = (V_0 - V_2) \times (V_3 - V_2)$$

$$\vec{n}_2 = (V_3 - V_1) \times (V_0 - V_1)$$

$$\vec{n}_3 = (V_2 - V_0) \times (V_1 - V_0)$$

法线为 $\vec{n}_i$ 的面就是与顶点 V_i 相对并且包含顶点 V_{3-i} 的面。

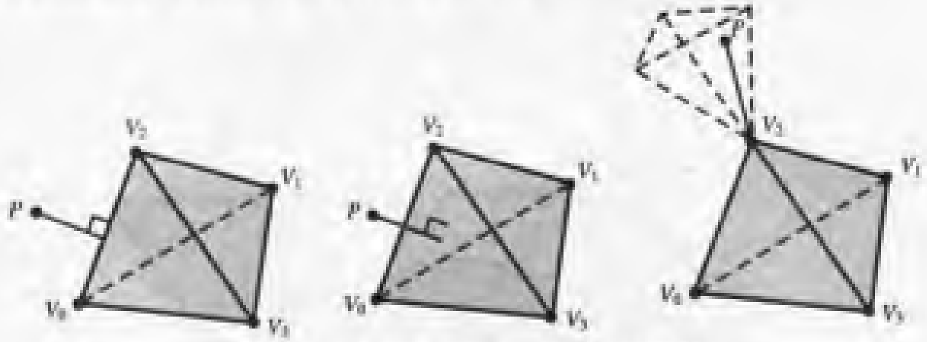


图 10.20 点到多面体（四面体）的距离

如果点 P 位于每一个面所在平面的反面，即如果对于所有的 i 都有 $\vec{n}_i \cdot (P - V_{3-i}) < 0$，那么它就位于四面体内。此时，距离为零。点 P 也可能位于四面体的边界上，此时对于所有的 i 都有 $\vec{n}_i \cdot (P - V_{3-i}) \leqslant 0$，并且至少有一个 i 使等号成立。如果等号只出现一次，那么点在面上，但并不在边或顶点上。如果等号出现两次，那么点在边上，但并不在顶点上。如果等号出现三次，那么点在顶点上。等号不可能出现四次。

我们知道，如果 P 不包含在四面体内，那么它与一个、两个或三个面之间的距离是非负的。因此，我们可计算点积

$$\vec{n}_i \cdot (P - V_{3-i})$$

其中 $i = 0, 1, 2, 3$。通过注意所有四个点积的符号，我们可以确定点是在哪个面（如果有的话）的外面。然后，我们就能计算点到每一个点积为负的面的距离，并取它们中的最小值作为我们的最终结果。

2. 点到凸多面体的距离

如果凸多面体是严格凸的（没有面共平面），那么可以采用一种类似于计算四面体的距离的方法。设凸面体的面都包含于平面 $\vec{n}_i \cdot (X - V_i) = 0$（其中 V_i 为一个顶点且 $\vec{n}_i$ 是指向面外的法线向量）内。当对于所有的 i 都有 $\vec{n}_i \cdot (P - V_i) < 0$ 时，点 P 就位于四面体内。当对于某些 i 有 $\vec{n}_i \cdot (P - V_i) > 0$ 时，点 P 就位于四面体外。如果对于所有的 i 都有 $\vec{n}_i \cdot (P - V_i) \leqslant 0$，且至少有一个 i 使等号成立，此时点 P 位于边界上。如果等号只出现一次，则点就在面内。如果等号出现两次，则点在边上。如果等号出现三次或三次以上，则点在顶点上。在最后一种情形中，等号出现的次数就是共用顶点的面的数目。

在任何情形中，如果我们依次将面标记为 $i = 0, 1, 2, \ldots, n-1$，并且所有点积都是负值，那么 P 在多面体内，且距离为零。否则，如果一个或多个面的有符号距离是非负的，那么我们就必须计算 P 与每一个这样的面的距离，并且 P 与多面体的距离就是这类距离中的最

小值。如果面是三角形的，那么可以利用 10.3.2 节中的算法。否则，必须使用 10.3.4 节中介绍的更一般的算法。

3. 点到一般多面体的距离

首先，我们知道多面体内的点与多面体的距离为零。因此，一般我们首先利用 13.4.3 节描述的算法来测试点是否在一般多面体内。如果点在多面体内，那么距离为零，并且不再需要做进一步的测试。否则，我们采用一种类似于前一节用于凸多面体的方法，来处理每一个面——如果点 P 在面外，则根据面是三角形还是非三角形，分别采用 10.3.2 节或 10.3.4 节中的算法来计算点与面的距离。

10.4.2 点到有向有界箱的距离

在本节中，我们将讨论计算点到有向有界箱（OBB）的距离的有关问题。一个有向有界箱可以用一个中心 C，构成一个右手正交基底的三个单位向量 $\hat{u}$，$\hat{v}$ 和 $\hat{w}$，以及三个表示 u，v，w 方向上的边长的一半的三个常数 h_u，h_v，h_w 来定义。如图 10.21 所示。

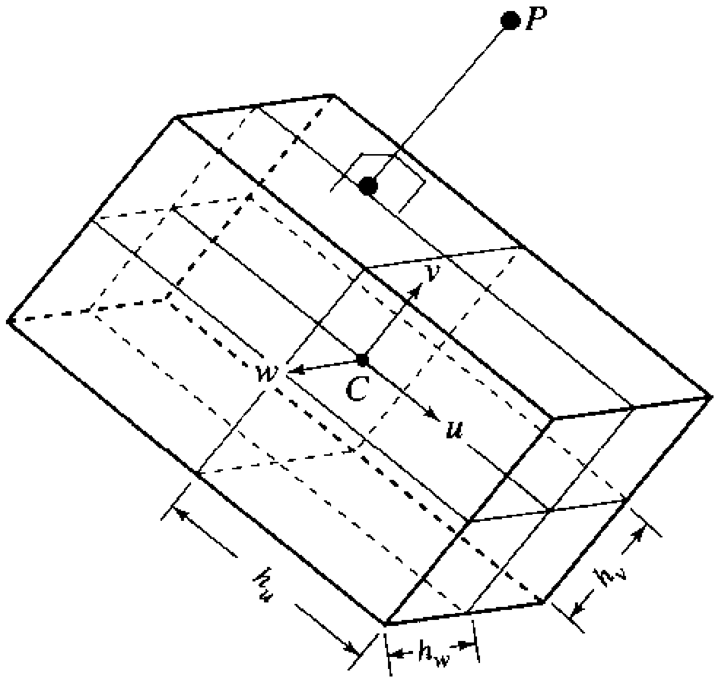


图 10.21　点到有向有界箱的距离

当然，我们也可以将有向有界箱看成是由 6 个矩形构成的，并简单地计算点到每一个面的距离，它们的最小值就是点到有向有界箱的距离。但是，这样做的效率很低。

有向有界箱的中心 C 和 $\hat{u}$，$\hat{v}$ 和 $\hat{w}$ 定义了一个坐标系。而且，相对于该坐标系，有向有界箱是中心位于原点的轴对齐箱，其面分别位于 XY，XZ 或 YZ 平面上。如果我们在该坐标系中计算点 P 与它的距离，那么我们就可以利用它的轴对齐、中心位于原点的性质，并计算最接近的点，以及该点与点 P 的距离。这很简单。

对于给定的有向有界箱的中心 C 和 $\hat{u}$，$\hat{v}$ 和 $\hat{w}$，点 P 在该坐标系中的坐标可以表示为

$$P' = [\,(P - C)\cdot\hat{u} \quad (P - C)\cdot\hat{v} \quad (P - C)\cdot\hat{w}\,]$$

也就是说，简单地将点 P 和 C 构成的向量投影到有向有界箱的每一个基底向量上。注

意，这在功能上与建立一个从全局空间到有向有界箱的局部的变换T与P的乘积等价，但效率更高。其变换T的变换矩阵为

$$\mathbf{T}=\begin{bmatrix}\hat{u}^{T} & \hat{v}^{T} & \hat{w}^{T} & \vec{0}^{T}\\ & -C & & 1\end{bmatrix}$$

因此，我们可以简单地将P'投影到这个箱上，这将得到与P'最接近的Q'，并可通过简单地计算得到它们之间的距离，这也是原来的点P与箱的距离。如果我们要求最接近的点在全局空间上的坐标，只需简单地用$\mathbf{T}^{-1}$来变换Q'，这将得到最接近于P的点Q。图 10.22 显示了这样的一个例子，为了表示的方便，图形显示在二维空间中。

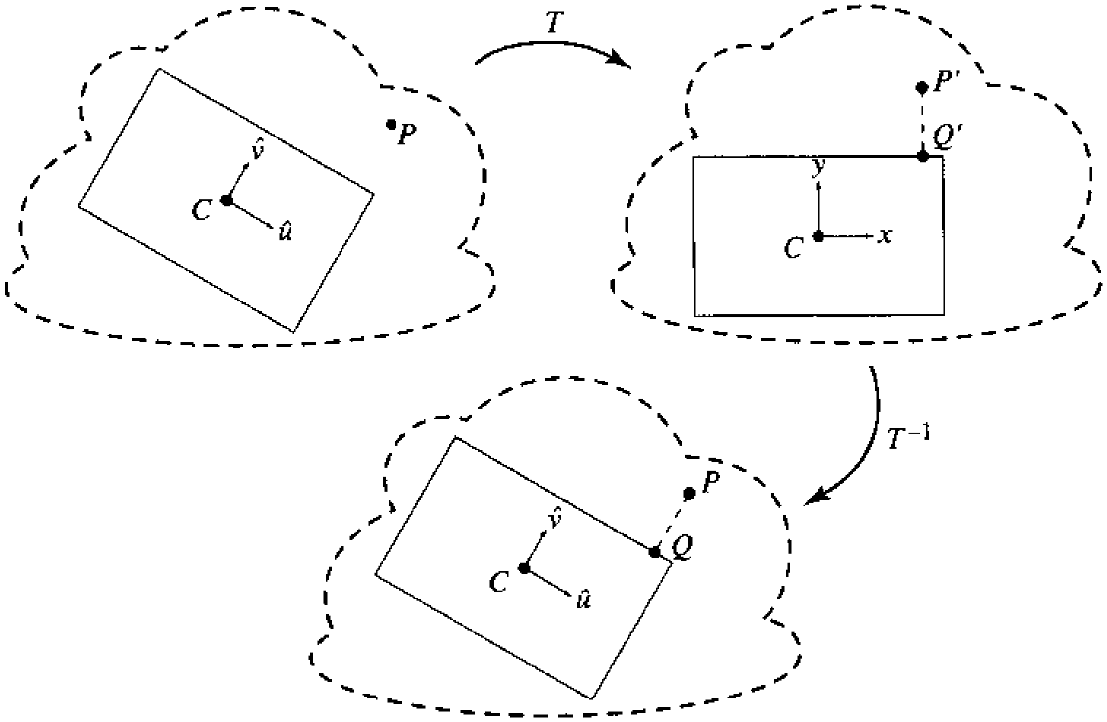


图 10.22　计算点与 OBB 的距离

伪码如下（注意，我们计算的是距离的平方，而且仅在要求时才返回最接近的点Q）：

```
float PointOBBDistanceSquared(Point p, OBB box, boolean computePoint,
                              Point& q)
{
    // Transform p into box's coordinate frame
    Vector offset = p - box.center();
    Point  pPrime(Dot(offset, box.u), Dot(offset, box.v), Dot(offset, box.w));

    // Project pPrime onto box
    float distanceSquared = 0;
    float d;

    if (pPrime.x < -box.uHalf) {
        d = pPrime.x + box.uHalf;
        distanceSquared += d * d;
        qPrime.x = -box.uHalf;
    } else if (pPrime.x > box.uHalf) {
```

```
            d = pPrime.x - box.uHalf;
            distanceSquared += d * d;
            qPrime.x = box.uHalf;
        } else {
            qPrime.x = pPrime.x;
        }

        if (pPrime.y < -box.vHalf ) {
            d = pPrime.y + box.v;
            distanceSquared += d * d;
            qPrime.y = -box.vHalf;
        } else if (pPrime.y > box.vHalf) {
            d = pPrime.y - box.vHalf;
            distanceSquared += d * d;
            qPrime.y = box.vHalf;
        } else {
            qPrime.y = pPrime.y;
        }

        if (pPrime.z < -box.wHalf) {
            d = pPrime.z + box.wHalf;
            distanceSquared += d * d;
            qPrime.z = -box.wHalf;
        } else if (pPrime.z > box.wHalf) {
            d = pPrime.z - box.wHalf;
            distanceSquared += d * d;
            qPrime.z = box.wHalf;
        } else {
            qPrime.z = pPrime.z;
        }

        //  If requested, compute the nearest point in global space (T^{-1})
        if (computePoint) {
            q.x = qPrime.x * box.u.x + qPrime.y * box.v.x + qPrime.z * box.w.x;
            q.y = qPrime.x * box.u.y + qPrime.y * box.v.y + qPrime.z * box.w.y;
            q.x = qPrime.x * box.u.z + qPrime.y * box.v.z + qPrime.z * box.w.z;

            q += box.center;
        }

        return distanceSquared;
    }
```

10.4.3　点到正交平截体的距离

本节的材料取自于 6.3.3 节中的二维情形。计算点到正交平截体的距离的算法的基础是确定正交平截体的面、边和顶点的 Voronoi 区域。给定一组对象，单个对象的 Voronoi 区域就是比其他任何对象都更接近于该对象的所有点的集合。需要计算包含这个点的 Voronoi

区域。该区域内的正交平截体上最近的点也要计算。基于这些计算就可求以得点到正交平截体的距离。

正交视图平截体（orthogonal view frustum）具有原点 E。其坐标轴由左向量 $\hat{l}$，顶向量 $\hat{u}$ 和方向向量 $\hat{d}$ 确定。按这种次序排列的这些向量构成一个右手正交系。该平截体在 $\hat{d}$ 方向上的扩展是 $[n, f]$，其中 $0 < n < f$。平截体在近平面上的四个角为 $E \pm \ell\hat{l} \pm \mu\hat{u} + n\hat{d}$。平截体在远平面上的四个角为 $E + (f/n)(\pm\ell\hat{l} \pm \mu\hat{u} + n\hat{d})$。平截体轴是原点为 E，方向为 $\hat{d}$ 的射线。我们说平截体是正交的，是因为它的轴垂直于近面和远面。

设 P 为需求与平截体的距离的点，该点可在平截体坐标系中表示为

$$P = E + x_0\hat{l} + x_1\hat{u} + x_2\hat{d}$$

因此 $x_0 = \hat{l} \cdot (P - E)$，$x_1 = \hat{u} \cdot (P - E)$ 且 $x_2 = \hat{d} \cdot (P - E)$。证明 $x_0 \geqslant 0$ 和 $x_1 \geqslant 0$ 的计算方法就足够了。其方法与二维空间的情形完全一样：反射 x_0 和 x_1 分量，寻找最近的点，然后将 x_0 和 x_1 分量反射回原象限。

平截体分量的命名惯例是近分量用 N 表示，远分量用 F 表示，上分量用 U 表示，左分量用 L 表示。平截体的顶面标记为 F 面。它有两条边：UF 边的方向为 $\hat{l}$，LF 边的方向为 $\hat{u}$。它还有一个顶点：位于 $(f\ell/n, f\mu/n, f)$ 的 LUF 顶点。平截体的底面标记为 N 面。它有两条边：UN 边的方向为 $\hat{l}$，LN 边的方向为 $\hat{u}$。它还有一个顶点：位于 (ℓ, μ, n) 的 LUN 顶点。其余的两个面为法线为 $(n, 0, -\ell)$ 的 L 面和法线为 $(0, n, -\mu)$ 的 U 面。最后，还有一条 L 面和 U 面共用的边 LU 边。图 10.23 说明了 Voronoi 区域的边界。细的黑线表示分隔一些 Voronoi 区域的近面和远面。

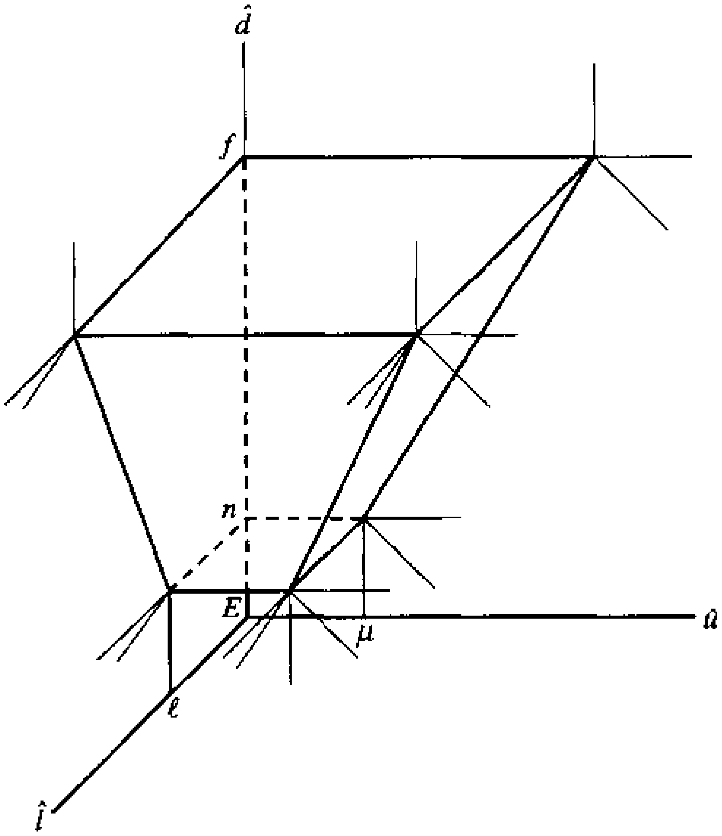


图 10.23　第一象限内平截体的分区

确定(x_0, x_1, x_2)的 Voronoi 区域的伪码列出如下：

```
if (x2 >= f) {
    if (x0 <= f * l / n) {
        if (x1 <= f * u / n)
            F-face is closest;
        else
            UF-edge is closest;
    } else {
        if (x1 <= f * u / n)
            LF-edge is closest;
        else
            LUF-vertex is closest;
    }
} else if (x2 <= n) {
    if (x0 <= l) {
        if (x1 <= u)
            N-face is closest;
        else {
            t = u * x1 + n * x2;
            if (t >= (f / n) * (u * u + n * n))
                UF-edge is closest;
            else if (t >= u * u + n * n)
                U-face is closest;
            else
                UN-edge is closest;
        }
    } else {
        if (x1 <= u) {
            t = l * x0 + n * x2;
            if (t >= (f / n) * (l * l + n * n))
                LF-edge is closest;
            else if (t >= l * l + n * n)
                L-face is closest;
            else
                LN-edge is closest;
        } else {
            r = l * x0 + u * x1 + n * x2;
            s = u * r - (l * l + u * u + n * n) * x1;
            if (s >= 0.0) {
                t = l * x0 + n * x2;
                if (t >= (f / n) * (l * l + n * n))
                    LF-edge is closest;
                else if (t >= l * l + n * n)
                    L-face is closest;
                else
                    LN-edge is closest;
            } else {
                s = l * r - (l * l + u * u + n * n) * x0;
                if (s >= 0.0) {
                    t = u * x1 + n * x2;
```

```
                    if (t >= (f / n) * (u * u + n * n))
                        UF-edge is closest;
                    else if (t >= u * u + n * n)
                        U-face is closest;
                    else
                        UN-edge is closest;
                } else {
                    if (r >= (f / n)(l * l + u * u + n * n))
                        LUF-vertex is closest;
                    else if (r >= l * l + u * u + n * n)
                        LU-edge is closest;
                    else
                        LUN-vertex is closest;
                }
            }
        }
    }
} else {
    s = n * x0 - l * x2;
    t = n * x1 - u * x2;
    if (s <= 0) {
        if (t <= 0)
            point inside frustum;
        else {
            t = u * x1 + n * x2;
            if (t >= (f / n) * (u * u + n * n))
                UF-edge is closest;
            else
                U-face is closest;
        }
    } else {
        if (t <= 0) {
            t = l * x0 + n * x2;
            if (t >= (f / n) * (l * l + n * n))
                LF-edge is closest;
            else
                L-face is closest;
        } else {
            r = l * x0 + u * x1 + n * x2;
            s = u * r - (l * l + u * u + n * n) * x1;
            if (s >= 0) {
                t = l * x0 + n * x2;
                if (t >= (f / n) * (l * l + n * n))
                    LF-edge is closest;
                else
                    L-face is closest;
            } else {
                t = l * r - (l * l + u * u + n * n) * x0;
```

```
                if (t >= 0) {
                    t = u * x1 + n * x2;
                    if (t >= (f / n) * (u * u + n * n))
                        UF-edge is closest;
                    else
                        U-face is closest;
                } else {
                    if (r >= l * l + u * u + n * n)
                        LUF-vertex is closest;
                    else
                        LU-edge is closest;
                }
            }
        }
    }
}
```

在每一个区域内的最近点可通过在该分量上的投影来获得。

10.5 点到二次曲面的距离

在本节中，我们将讨论计算点到二次曲面的距离的问题。我们将看到，一般情形需要计算六次方程：为了求得确切的最小距离，必须计算所有的根，并比较与它们相关的距离。我们可以利用特殊类型的二次曲面的几何性质，并获得相对高效的算法。我们给出了一个关于椭球面的例子。

10.5.1 点到一般二次曲面的距离

本节描述了一个计算点到二次曲面的距离的算法。一般二次曲面的方程为

$$Q(X) = X^{\mathrm{T}}\mathrm{A}X + B^{\mathrm{T}}X + c = 0$$

其中 A 是一个 3×3 的对称矩阵，不一定是可逆的（例如，当曲面是柱面和抛物面时）；B 是一个 3×1 的向量，c 是一个数量。参数 X 是一个 3×1 的向量。给定隐式定义为 $Q(X)=0$ 的曲面和一个点 Y，要求 Y 到曲面的距离以及最接近的点 X。

从几何意义上来说，曲面上最接近于 Y 的点 X 必须满足如下条件：$Y-X$ 是曲面的法线。由于曲面梯度 $\nabla Q(X)$ 是曲面的法线，因此最接近的点需要满足的代数条件为

$$Y - X = t\nabla Q(X) = t(2\mathrm{A}X + B)$$

其中 t 为数量。因此

$$X = (\mathrm{I} + 2t\mathrm{A})^{-1}(Y - tB)$$

其中 I 为单位矩阵。我们可将该方程代入一般二次曲面方程中的 X，并得到一个关于 t 的最高次数为 6 的多项式。

我们先进行可以简化编码的工作，而不是直接代入 X。用特征值分解来对 A 进行因式分解，可得 $\mathrm{A} = \mathrm{R}\mathrm{D}\mathrm{R}^{\mathrm{T}}$，其中 R 为一个正交矩阵，其各列为 A 的特征向量；D 为对角线矩阵，

其对角线元素为A的特征值。因此

$$\begin{aligned}X &= (\mathrm{I}+2t\mathrm{A})^{-1}(Y-tB)\\ &= (\mathrm{RR}^{\mathrm{T}}+2t\mathrm{RDR}^{\mathrm{T}})^{-1}(Y-tB)\\ &= [\mathrm{R}(\mathrm{I}+2t\mathrm{D})\mathrm{R}^{\mathrm{T}}]^{-1}(Y-tB)\\ &= \mathrm{R}(\mathrm{I}+2t\mathrm{D})^{-1}\mathrm{R}^{\mathrm{T}}(Y-tB)\\ &= \mathrm{R}(\mathrm{I}+2t\mathrm{D})^{-1}(\vec{\alpha}-t\vec{\beta})\end{aligned}$$

其中最后一个方程定义了 $\vec{\alpha}$ 和 $\vec{\beta}$。将其代入二次曲面方程，可得

$$0=(\vec{\alpha}-t\vec{\beta})^{\mathrm{T}}(\mathrm{I}+2t\mathrm{D})^{-1}\mathrm{D}(\mathrm{I}+2t\mathrm{D})^{-1}(\vec{\alpha}-t\vec{\beta})+\vec{\beta}^{\mathrm{T}}(\mathrm{I}+2t\mathrm{D})^{-1}(\vec{\alpha}-t\vec{\beta})+C$$

对角线矩阵的逆矩阵为

$$(\mathrm{I}+2t\mathrm{D})^{-1}=\mathrm{Diag}\{1/(1+2td_0),1/(1+2td_1),1/(1+2td_2)\}$$

两边乘以$((1+2td_0)(1+2td_1)(1+2td_2))^2$，可得一个最高次数为 6 的多项式方程。

计算该多项式方程的根，并对每一个根 t 计算$X=(\mathrm{I}+2t\mathrm{A})^{-1}(Y-tB)$。计算$X$与$Y$之间的距离，其中的最小值就是要求的点到曲面的距离。

10.5.2　点到椭球面的距离

上一节讨论了计算点到二次曲面的距离的一般方法。计算相交的方程最高可能是一个六次方程，可以计算这样的方程，但却非常费时。

对于特殊类型的二次曲面，可利用曲面的几何性质来简化计算。我们将以椭球面为例来加以说明。椭球面的方程为

$$q(\mathbf{x})=\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}-1=0 \tag{10.7}$$

设P为要求到曲面的距离的点。椭球面上与P最接近的点可以看成是经过P的法线与曲面的交点，即

$$P-\mathbf{x}=\nabla q(\mathbf{x}) \tag{10.8}$$

其中$\mathbf{x}$为曲面上的点，$\nabla q(\mathbf{x})$为经过$\mathbf{x}$的曲面的法线。

经过椭球体上一点的法线为

$$\begin{aligned}\nabla q(\mathbf{X}) &= (\frac{\partial q}{\partial x},\frac{\partial q}{\partial y},\frac{\partial q}{\partial z})\\ &= 2(\frac{x}{a^2},\frac{y}{b^2},\frac{z}{c^2})\end{aligned} \tag{10.9}$$

将方程（10.9）代入方程（10.8），并将结果代入方程（10.7），可得我们用于计算最接近的点的方程

$$a^2(\lambda+b^2)^2(\lambda+c^2)^2P_x^2+b^2(\lambda+a^2)^2(\lambda+c^2)^2P_y^2+c^2(\lambda+a^2)^2(\lambda+b^2)^2P_z^2$$
$$-(\lambda+a^2)^2(\lambda+b^2)^2(\lambda+c^2)^2=0$$

可以看出，这是一个六次方程。考虑消去λ，并使用两个未知数，比如，用常数和z来

表示x和y。这将得到一个令人难受的方程:

$$z^6(b-c)^2(b+c)^2(a-c)^2(a+c)^2+z^5 2c^2P_z(b-c)(b+c)(a-c)(a+c)(b^2+a^2-2c^2)$$
$$+z^4-c^2(-2a^4b^2c^2-6c^6P_z^2+6c^4P_z^2b^2+a^4b^4-a^4c^2P_z^2+c^4b^4$$
$$-2c^6b^2-2a^2c^2b^4-4c^2a^2P_z^2b^2-2a^2c^6+6c^4a^2P_z^2+c^8+a^4c^4$$
$$+4a^2c^4b^2+2c^2a^2P_x^2b^2-c^4P_y^2b^2-a^4P_y^2b^2-c^2P_z^2b^4-b^4P_x^2a^2$$
$$-c^4a^2P_x^2+2c^2b^2P_y^2a^2)+z^3-2c^4P_z(-c^2b^4+c^2P_y^2b^2+2c^4P_z^2$$
$$-4a^2b^2c^2+3b^2c^4-2c^6-c^2P_z^2b^2-b^2P_x^2a^2+3c^4a^2-a^2P_y^2b^2$$
$$+b^2a^4-a^4c^2-c^2a^2P_z^2+c^2a^2P_x^2+b^4a^2)+z^2-c^6P_z^2(4a^2b^2$$
$$-6a^2c^2-c^2P_z^2-6b^2c^2-P_y^2b^2-P_x^2a^2+6c^4+a^4+b^4)$$
$$+z^1-2c^8P_z^3(b^2+a^2-2c^2)+z^0-c^10P_z^4=0$$

另一种方法是，考虑如果P位于椭球面的中心的情形，如图 10.24 所示。我们必须考虑 6 个可能的最接近的点，其中至少两个将具有最小距离。

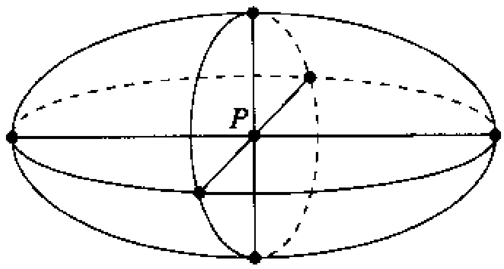


图 10.24 椭球面上 6 种可能的"最近点"

对于任意情形，Hart（1994）列出了如下的观察结果:

- λ 的图形在最大的根之外具有递减的坡度。这说明可以避免费时的六次方程的数值根求解过程（比如牛顿迭代法）。在这种情形中，提供一个"足够大"的初始估计值将导致快速的收敛。
- 如果P（确实）在曲面上，或者在椭球面内，初始估值$\lambda=0$将满足要求，因为最大的根将小于或等于零。
- 如果P在椭球面外，那么初始值

$$\lambda=\|P-\mathcal{O}\|\max\{a,b,c\}$$

将"足够大"（$\mathcal{O}$为原点）。

10.6 点到多项式曲线的距离

本节将讨论计算三维空间中点到多项式曲线的距离。不失一般性，我们假定曲线为参数形式的多项式曲线（例如，贝塞尔曲线），并可被分段表示（例如，NURBS）。

给定参数形式的曲线$Q(t)$和一个点P，我们要求在Q上最接近于P的点。也就是说，我们希望找到满足$Q(t)$与P之间距离为最小值的参数t。我们的方法的基础是，注意到从P到$Q(t)$的线段（我们希望其长度为最小值）垂直于与曲线相切于$Q(t_0)$的切线，如图 10.25 所示。

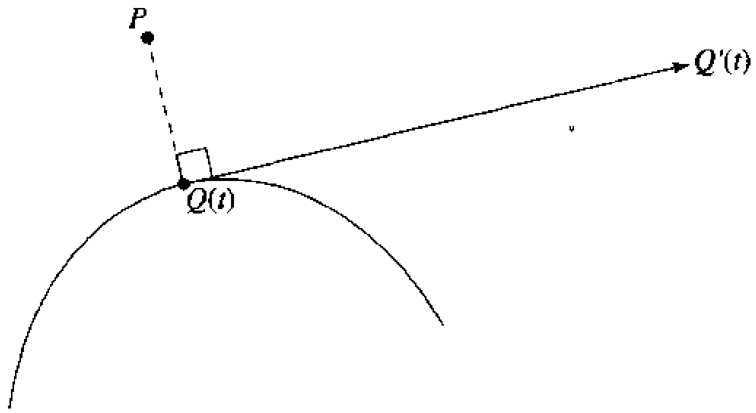


图 10.25 任意点到参数形式曲线的距离

我们希望求解的关于t的方程为

$$(Q(t)-P)\cdot Q'(t)=0 \tag{10.10}$$

如果$Q(t)$的次数为d，那么$Q'(t)$的次数为$d-1$，因此方程 10.10 的次数为$2d-1$。因此，对于任何次数高于 2 的曲线，不存在封闭形式的解，必须使用某些数值根计算方法。一种选择是使用牛顿迭代法。然而，这种方法要求有一个合理的初始估值。Piegl 和 Tiller（1995）建议的一种方法是评估曲线上位于n个等距分布的参数值处的点（或者分段多项式曲线上位于n个等距分布的参数值处的点），并分别计算它们与P的距离（平方）。最接近的评估点的参数值可作为牛顿迭代法的初始估值。

假设我们有初始估值t_0。将从第i个牛顿迭代点处的参数值称为t_i。于是，牛顿步为

$$\begin{aligned}t_{i+1}&=t_i-\frac{(Q(t)-P)\cdot Q'(t)}{((Q(t)-P)\cdot Q'(t))'}\\&=t_i-\frac{(Q(t)-P)\cdot Q'(t)}{(Q(t)-P)\cdot Q''(t_i)+\|Q'(t_i)\|^2}\end{aligned} \tag{10.11}$$

当满足一定条件时，牛顿迭代一般是不连续的。Piegl 和 Tiller（1995）使用了两个零容许：

- ϵ_1：欧几里得距离测量
- ϵ_2：零余弦测量

它们按如下次序来检测条件：

（1）重合点：

$$\|Q(t_i)-P\|\leqslant\epsilon_1$$

（2）零余弦（$Q(t_i)-P$与$Q'(t)$之间的角度足够接近于 90°）：

$$\frac{\|(Q(t)-P)\cdot Q'(t)\|}{\|Q(t)-P\|\|Q'(t)\|}\leqslant \epsilon_2$$

如果这些条件中的任一个不满足，那么进行牛顿步。这样，检测另外两个条件：

（3）参数是否位于范围$a\leqslant t_i\leqslant b$之间

（4）参数是否没有显著的改变：

$$\|(t_{i+1}-t_i)Q'(t_i)\|\leqslant \epsilon_1$$

如果满足最后的条件，那么牛顿迭代是不连续的。当前的参数值t_{i+1}就是理想的根，最接近于P的点就是$Q(t_{i+1})$，从P到$Q(t)$的距离就是$\|(Q(t_{i+1})-P)\|$。

另一种方法，是将曲线转化为贝塞尔曲线的形式，并用 Schneider（1990）描述的基于贝塞尔曲线的求根方法。这种方法避免了为获得用于牛顿迭代法的合理的初始估值，而要求计算$Q(t)$上的一系列点的步骤。如果曲线已经具有贝塞尔曲线的形式，那么这种方法特别有效。

10.7 点到多项式曲面的距离

本节我们将介绍计算点到多项式曲面的距离问题。不失一般性，我们假设曲面是参数形式的多项式曲面（例如，贝塞尔曲面），并且可被分段（例如，NURBS 曲面）表示。

给定一个参数曲面$S(u,v)$和一个点P，我们要求曲面S上最接近于P的点。也就是说，我们希望找到满足$S(u,v)$与P的距离最小的参数(u,v)。我们的方法的基础是，注意到从P到$Q(t)$的线段（我们希望其长度为最小值）垂直于与曲面相切于$S(u_0,v_0)$的切面，如图 10.26 所示。

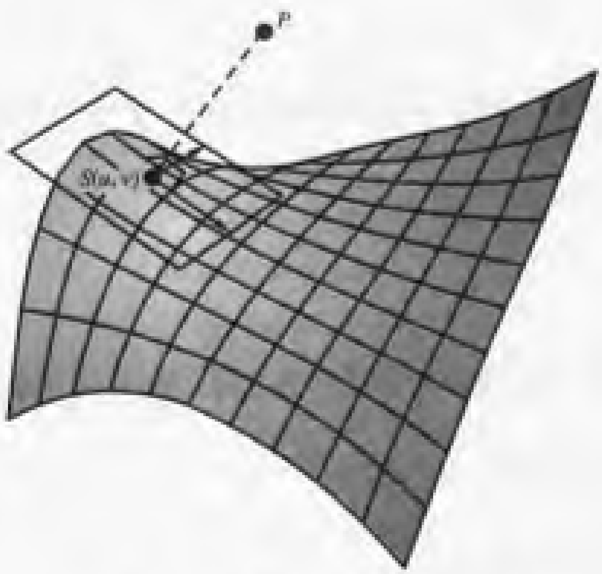


图 10.26 任意点到参数形式曲面的距离

曲面与任意点之间的向量可表示为参数形式曲面的一个函数：

$$r(u,v)=S(u,v)-P$$

直线$S(u,v)-P$与切面垂直的前提条件包括两个条件，即直线必须在每一个方向上垂直于偏微分（$S_u(u,v)$和$S_v(u,v)$）：

$$\begin{aligned} f(u,v) &= r(u,v)\cdot S_u(u,v) \\ &= 0 \\ g(u,v) &= r(u,v)\cdot S_v(u,v) \\ &= 0 \end{aligned}$$

因此，为了找到最接近的点，我们必须求解该方程系统。与我们在寻找点到多项式曲线的距离时遇到的问题相似，我们采用牛顿迭代法。同样，通过评估曲面上以$n\times n$规则分布的点，我们将发现一个初始估值。计算每一个点与P的距离（平方），并以最接近点的参数(u,v)作为牛顿迭代法的初始估值。

设

$$\begin{aligned} \sigma_i &= \begin{bmatrix} \delta u \\ \delta v \end{bmatrix} \\ &= \begin{bmatrix} u_{i+1}-u_i \\ v_{i+1}-v_i \end{bmatrix} \end{aligned}$$

$$\begin{aligned} J_i &= \begin{bmatrix} f_u(u_i,v_i) & f_v(u_i,v_i) \\ g_u(u_i,v_i) & g_v(u_i,v_i) \end{bmatrix} \\ &= \begin{bmatrix} \|S_u(u_i,v_i)\|^2 + r(u_i,v_i)\cdot S_{uu}(u_i,v_i) & S_u(u_i,v_i)\cdot S_v(u_i,v_i) + r(u_i,v_i)\cdot S_{uv}(u_i,v_i) \\ S_u(u_i,v_i)\cdot S_v(u_i,v_i) + r(u_i,v_i)\cdot S_{vu}(u_i,v_i) & \|S_v(u_i,v_i)\|^2 + r(u_i,v_i)\cdot S_{vv}(u_i,v_i) \end{bmatrix} \end{aligned}$$

$$\kappa_i = -\begin{bmatrix} f(u_i,v_i) \\ g(u_i,v_i) \end{bmatrix}$$

假设我们有一个初始估值(u_0,v_0)。在第i牛顿迭代步中，求解σ_i中2×2的方程系统：

$$J_i\sigma_i = \kappa_i$$

并计算参数值如下

$$u_{i+1} = \delta u + u_i$$

$$v_{i+1} = \delta v + v_i$$

当满足一定条件时，牛顿迭代一般是不连续的。Piegl 和 Tiller（1995）使用了两个零容许：

- ϵ_1：欧几里得距离测量
- ϵ_2：零余弦测量

它们按如下次序来检测条件：

（1）重合点：

$$\|S(u_i,v_i)-P\| \leq \epsilon_1$$

（2）零余弦（$S(u_i,v_i)-P$与$S_u(u_i,v_i)$之间的角度足够接近于 90°）对于$S_v(u_i,v_i)$，情况与此类似：

$$\frac{\|S_u(u_i,v_i)\cdot(S(u_i,v_i)-P)\|}{\|S_u(u_i,v_i)\|\|S(u_i,v_i)-P\|}\leqslant\epsilon_2$$

$$\frac{\|S_v(u_i,v_i)\cdot(S(u_i,v_i)-P)\|}{\|S_v(u_i,v_i)\|\|S(u_i,v_i)-P\|}\leqslant\epsilon_2$$

如果这些条件中的任意一个不满足，那么进行牛顿步。这样，检测另外两个条件：

（3）参数是否位于范围 $a\leqslant u_i\leqslant b$ 和 $c\leqslant v_i\leqslant d$ 之间

（4）参数是否没有显著的改变：

$$\|(u_{i+1}-u_iS_u(u_i,v_i))+(v_{i+1}-v_iS_u(u_i,v_i))\|\leqslant\epsilon_2$$

10.8　线形对象之间的距离

在本节中，我们将讨论计算线形对象之间的距离的问题，包括直线、线段和射线的各种组合之间的距离。

10.8.1　直线与直线之间的距离

假设有两条直线 $\mathcal{L}_0(s)=P_0+s\vec{d}_0$ 和 $\mathcal{L}_1(t)=P_1+t\vec{d}_1$，我们希望找到它们之间的最小距离。设 $Q_0=P_0+s_c\vec{d}_0$ 和 $Q_1=P_1+t_c\vec{d}_1$ 为分别位于 $\mathcal{P}_0$ 和 $\mathcal{P}_1$ 处的点，它们之间的距离是最小距离，并设 $\vec{v}=Q_0-Q_1$（如图 10.27 所示）。

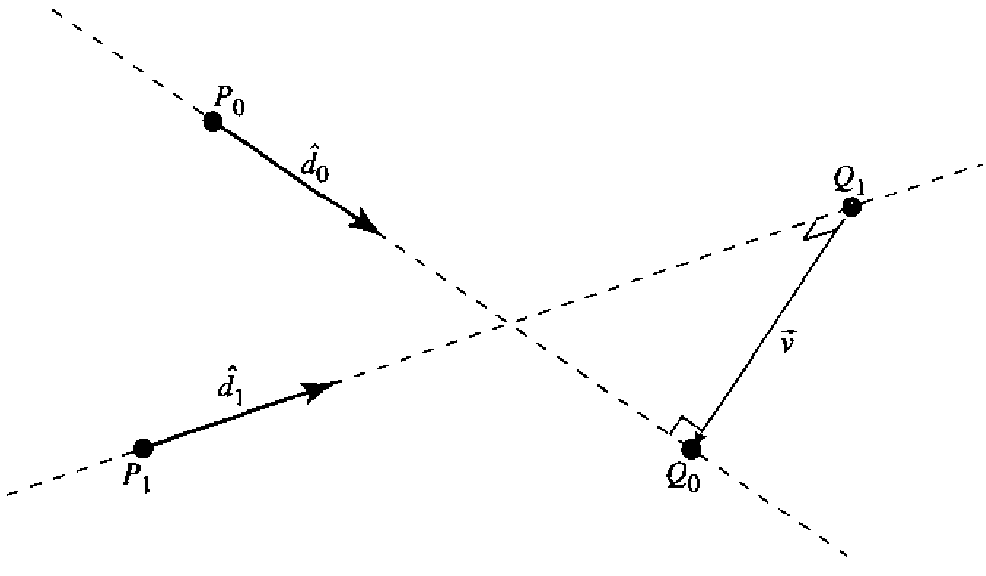


图 10.27　两条直线之间的距离

解决寻找 s_c 和 t_c（并据此计算最小距离）的问题的关键，是注意到仅当 $\|\vec{v}\|$ 具有最小值时，$\vec{v}$ 才同时垂直于 $\mathcal{L}_0$ 和 $\mathcal{L}_1$。用数学公式来表示，就是必须同时满足

$$\vec{d}_0\cdot\vec{v}=0 \tag{10.12}$$

$$\vec{d}_1\cdot\vec{v}=0 \tag{10.13}$$

如果我们扩展 $\vec{v}$ 的定义式，

$$\vec{v} = Q_0 - Q_1$$
$$= P_0 + s_c\vec{d}_0 - P_1 + t_c\vec{d}_1$$

并将其代回方程（10.12）和方程（10.13），可得

$$(\vec{d}_0 \cdot \vec{d}_0)s_c - (\vec{d}_0 \cdot \vec{d}_1)t_c = -\vec{d}_0 \cdot (P_0 - P_1)$$
$$(\vec{d}_1 \cdot \vec{d}_0)s_c - (\vec{d}_1 \cdot \vec{d}_1)t_c = -\vec{d}_1 \cdot (P_0 - P_1)$$

设$a = \vec{d}_0 \cdot \vec{d}_0$，$b = -\vec{d}_0 \cdot \vec{d}_1$，$c = \vec{d}_1 \cdot \vec{d}_1$，$d = \vec{d}_0 \cdot (P_0 - P_1)$，$e = \vec{d}_1 \cdot (P_0 - P_1)$，且$f = (P_0 - P_1) \cdot (P_0 - P_1)$。我们得到两个具有两个未知数的方程，其解为

$$s_c = \frac{be - cd}{ac - b^2}$$
$$t_c = \frac{bd - ae}{ac - b^2}$$

如果分母$ac - b^2 < \epsilon$，那么$\mathcal{L}_0$和$\mathcal{L}_1$互相平行。在这种情形中，我们可以任意选取t_c的值来计算s_c。通过设$t_c = 0$，可以将计算量减到最小程度，此时仅需计算$s_c = -d/a$。

最后，一旦我们计算得到了最接近的点的参数值，就能计算$\mathcal{L}_0$和$\mathcal{L}_1$之间的距离：

$$\|\mathcal{L}_0(s_c) - \mathcal{L}_1(t_c)\| = \|(P_0 - P_1) + \frac{(be - cd)\vec{d}_0 - (bd - ae)\vec{d}_1}{ac - b^2}\|$$

伪码为

```
float LineLineDistanceSquared(Line line0, Line line1)
{
    u = line0.base - line1.base;
    a = Dot(line0.direction, line0.direction);
    b = Dot(line0.direction, line1.direction);
    c = Dot(line1.direction, line1,direction);
    d = Dot(line0.direction, u);
    e = Dot(line1.direction, u);
    f = Dot(u, u);
    det = a * c - b * b;

    // Check for (near) parallelism
    if (det < epsilon) {
        // Arbitrarily choose the base point of line0
        s = 0;
        // Choose largest denominator to minimize floating-point problems
        if (b > c) {
            t = d / b;
        } else {
            t = e / c;
        }
        return d * s + f;
    } else {
        // Nonparallel lines
        invDet = 1 / det;
        s = (b * e - c * d) * invDet;
        t = (a * e - b * d) * invDet;
```

```
        return s * (a * s + b * t + 2 * d) + t * (b * s + c * t + 2 * e) + f;
    }
}
```

10.8.2 线段 / 线段、直线 / 射线、直线 / 线段、射线 / 射线、射线 / 线段之间的距离

有三种不同的线形对象，即直线、射线和线段，它们将产生 6 种不同的距离测试组合（线段/线段、直线/射线、直线/线段、射线/射线、射线/线段，以及直线/直线）。

让我们先回顾一下已知的知识，重新考虑学过的数学问题。正如我们刚刚看到的一样，寻找两条直线之间的距离等价于计算满足向量$\vec{v} = Q_1 - Q_0$的长度为最小值的 s 和 t。我们可以将其改写为

$$
\begin{aligned}
\|\vec{v}\|^2 &= \vec{v} \cdot \vec{v} \\
&= ((P_0 - P_1) + s\vec{d}_0 - t\vec{d}_1) \cdot ((P_0 - P_1) + s\vec{d}_0 - t\vec{d}_1)
\end{aligned}
$$

这是一个关于 s 和 t 的二次函数，即它是一个形状为抛物面的函数$f(s,t)$。对于直线的情形，s 和 t 的定义域没有限制，并且它的解$[s_c,t_c]$对应于f具有最小值的点（即抛物面的"底"）。

然而，如果其中的一条线形对象为射线或线段，那么 s 和 t 的定义域就有限制了。对于射线的情形，s（或 t）必须为非负的；对于线段的情形，$0 \leqslant s \leqslant 1$（$t$ 也与此类似）。我们可以建立一个表来表示所有可能组合的定义域限制（如图 10.28 所示）。

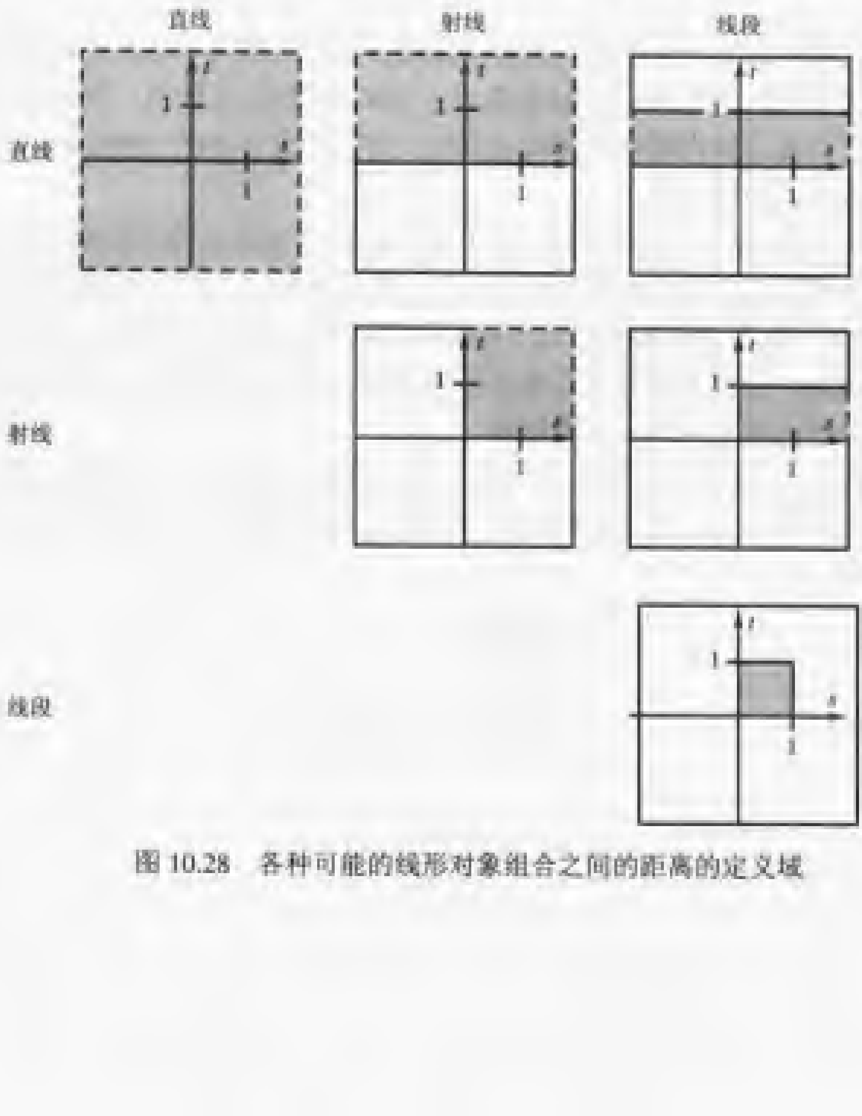


图 10.28 各种可能的线形对象组合之间的距离的定义域

一般地，二次函数的全局最小值不可能在限制的定义域之内出现，在这种情形中，最小值将出现在沿着定义域的某一条边的某个点处。其中一条线形对象的参数值的限制所产生的定义域分区可用来生成一个算法，用来对(s_c, t_c)的位置进行分类，并用于一组特定的操作，以确定确切的最接近的点，以及它们之间的距离（10.8.3 节显示了这种方法用于确定两条线段之间的相交的过程）。然而，也可以使用 Dan Sunday(2001b)设计的一种在某种程度上更简单的方法。

类似于对(s_c, t_c)所在的区域进行分类的方法，这种方法考虑有界定义域的哪一条边对(s_c, t_c)是“可见的”。例如，对于求线段／线段之间距离的情形，定义域限制在$[0, 1] \times [0, 1]$内。图 10.29 显示了一个解的两种可能的可见性条件：对于左图，只有边界$t = 1$是可见的；对于右图，$s = 1$和$t = 1$都是可见的。

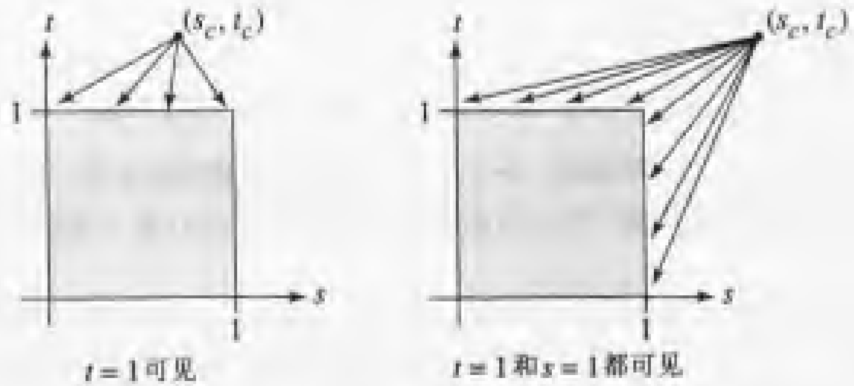


图 10.29　定义域边界可见性的定义

通过简单地比较s_c和t_c，我们可以很容易地确定定义域的哪一条边界边是可见的。对于每一条可见的边界边，我们可以计算这条边上与(s_c, t_c)最接近的点。如果仅有一条边界边是可见的，那么最接近的点将位于区间$[0, 1]$内；否则，最接近的点将在该区域之外，因此，我们需要检查其他的可见边。

这种方法的基本思想是，首先计算射线或线段所在的无限直线上的最接近点，即s_c和t_c。如果这两个值都分别在线形对象的参数s和t的定义域之内，那么我们就已得到需要的结果。然而，如果一条或两条线形对象不是无限直线，而是射线或线段，那么s和（或t）的定义域是有限制的，而且我们必须计算在受限的定义域上使距离平方取得最小值的点。这些点将具有对应于边界上这些点的参数值。

在文献 Sunday(2001b)中，我们看到，利用一点点微积分方法，就可以很容易地计算边界边上的最接近点。对于边$s = 0$的情形，我们有$\|\vec{v}\|^2 = (\vec{u} - t\vec{d}_1) \cdot (\vec{u} - t\vec{d}_1)$，其中$\vec{u} = P_0 - P_1$。该式对$t$的导数为

$$0 = \frac{d}{dt}\|\vec{v}\|^2$$

$$= -2\vec{d}_1 \cdot (\vec{u} - t\vec{d}_1)$$

其最小值出现在

$$t' = \frac{\vec{d}_1 \cdot \vec{u}}{\vec{d}_1 \cdot \vec{d}_1}$$

如果 $0 \leqslant t' \leqslant 1$，那么这就是要求的确定解。否则，如果$t' > 1$，则确切解为 1，如果$t' < 0$，则确切解为 0。

边 $t=0$的情形完全与此类似：此时我们有$\|\vec{v}\|^2 = (-\vec{u} - s\vec{d}_0) \cdot (-\vec{u} - s\vec{d}_0)$。该式对 s 的导数为

$$0 = \frac{d}{ds}\|\vec{v}\|^2$$
$$= -2\vec{d}_0 \cdot (-\vec{u} - s\vec{d}_0)$$

其最小值出现在

$$s' = \frac{-\vec{d}_0 \cdot \vec{u}}{\vec{d}_0 \cdot \vec{d}_0}$$

如果 $0 \leqslant s' \leqslant 1$，那么这就是要求的确定解。否则，如果$s' > 1$，则确切解为 1；如果$s' < 0$，则确切解为 0。

对于边$s=1$的情形，我们有$\|\vec{v}\|^2 = (\vec{u} + \vec{d}_0 - t\vec{d}_1) \cdot (\vec{u} + \vec{d}_0 - t\vec{d}_1)$。该式对 t 的导数为

$$0 = \frac{d}{dt}\|\vec{v}\|^2$$
$$= -2\vec{d}_1(\vec{u} - t\vec{d}_1 + \vec{d}_0)$$

其最小值出现在

$$t' = \frac{\vec{d}_1 \cdot \vec{u} + \vec{d}_0 \cdot \vec{d}_1}{\vec{d}_1 \cdot \vec{d}_1}$$

如果 $0 \leqslant t' \leqslant 1$，那么这就是要求的确定解。否则，如果$t' > 1$，则确切解为 1，如果$t' < 0$，则确切解为 0。

对于边$t=1$的情形，我们有$\|\vec{v}\|^2 = (-\vec{u} + \vec{d}_1 - s\vec{d}_0) \cdot (-\vec{u} + \vec{d}_1 - s\vec{d}_0)$。该式对 s 的导数为

$$0 = \frac{d}{ds}\|\vec{v}\|^2$$
$$= -2\vec{d}_0(-\vec{u} - s\vec{d}_0 + \vec{d}_1)$$

其最小值出现在

$$s' = \frac{-\vec{d}_0 \cdot \vec{u} + \vec{d}_1 \cdot \vec{d}_0}{\vec{d}_0 \cdot \vec{d}_0}$$

如果 $0 \leqslant s' \leqslant 1$，那么这就是要求的确定解。否则，如果 $s' > 1$，则确切解为 1，如果$s' < 0$，则确切解为 0。

图 10.30 应该能更清楚地说明这一点，但应该注意，图形只是示意性的。并不要求两个线形对象必须垂直。

1．线段到线段的距离

图 10.31 显示了要求它们之间的距离的两条线段，以及其解的有限制定义域。在这种情形中，解的定义域限制在$[0,1] \times [0,1]$范围内。定义域由 4 条边所包围，即 $s=0$，$s=1$，$t=0$及$t=1$。如果s_c 和 t_c 中有一个位于该区域之外，那么我们就必须在定义域的边界边

上寻找最接近于点(s_c, t_c)的点。定义域虽然由 4 条边所包围，但是很清楚，我们最多只需在两条边上寻找边界点。

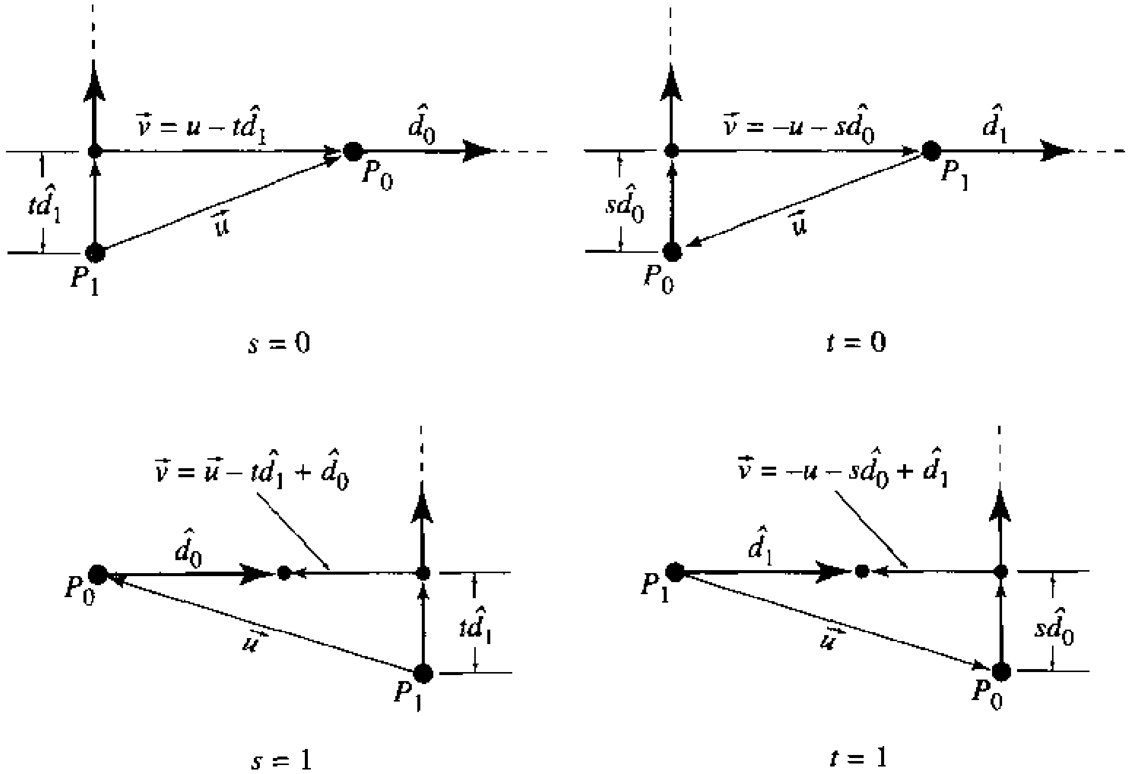


图 10.30　定义域有 4 条边的各种情形

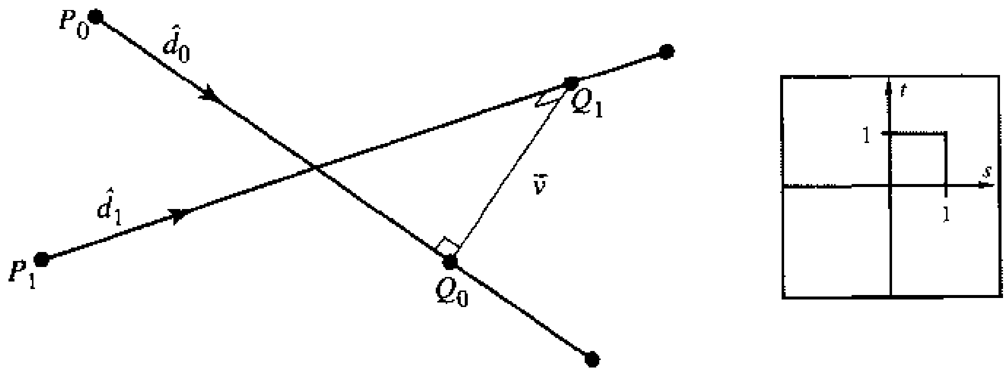


图 10.31　两条线段之间的距离

伪码为

```
float SegmentSegmentDistance3D(Segment seg0, Segment seg1)
{
    u = seg0.base - seg1.base;
    a = Dot(seg0.direction, seg0.direction);
    b = Dot(seg0.direction, seg1.direction);
    c = Dot(seg1.direction, seg1.direction);
    d = Dot(seg0.direction, u);
    e = Dot(seg1.direction, u);
```

```
det = a * c - b * b;

// Check for (near) parallelism
if (det < epsilon) {
    // Arbitrary choice
    sNum = 0;
    tNum = e;
    tDenom = c;
    sDenom = det;
} else {
    // Find parameter values of closest points
    // on each segment's infinite line. Denominator
    // assumed at this point to be ''det'',
    // which is always positive. We can check
    // value of numerators to see if we're outside
    // the [0, 1] x [0, 1] domain.
    sNum = b * e - c * d;
    tNum = a * e - b * d;
}

// Check s
sDenom = det;
if (sNum < 0) {
    sNum = 0;
    tNum = e;
    tDenom = c;
} else if (sNum > det) {
    sNum = det;
    tNum = e + b;
    tDenom = c;
} else {
    tDenom = det;
}

// Check t
if (tNum < 0) {
    tNum = 0;
    if (-d < 0) {
        sNum = 0;
    } else if (-d > a) {
        sNum = sDenom;
    } else {
        sNum = -d;
        sDenom = a;
    }
} else if (tNum > tDenom) {
    tNum = tDenom;
    if ((-d + b) < 0) {
        sNum = 0;
    } else if ((-d + b) > a) {
```

```
            sNum = sDenom;
        } else {
            sNum = -d + b;
            sDenom = a;
        }
    }

    // Parameters of nearest points on restricted domain
    s = sNum / sDenom;
    t = tNum / tDenom;

    // Dot product of vector between points is squared distance
    // between segments
    v = seg0.base + (s * seg0.direction) - seg1.base + (t * seg1.direction);
    return Dot(v,v);
}
```

2. 直线到射线的距离

图 10.32 显示了要求它们之间的距离的一条射线和一条直线，以及其解的有限制定义域。在这种情形中，距离函数的定义域为$[-\infty,\infty]\times[0,\infty)$（或相反）。定义域仅由一条边所包围，对应于$s=0$或$t=0$。如果在射线所在的无限直线上的最接近点的参数小于 0，那么我们就必须在一条边上寻找最接近的点。

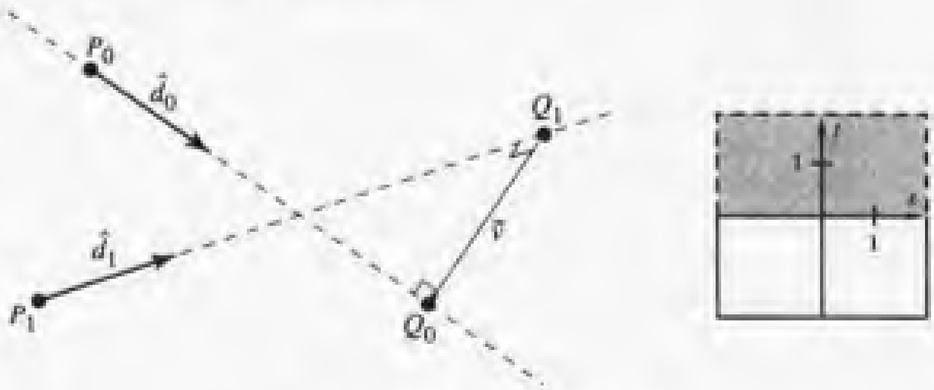


图 10.32　直线与射线之间的距离

伪码为

```
float LineRayDistance3D(Line line, Ray ray)
{
    u = line.base - ray.base;
    a = Dot(line.direction, line.direction);
    b = Dot(line.direction, ray.direction);
    c = Dot(line.direction, ray.direction);
    d = Dot(line.direction, u);
    e = Dot(ray.direction, u);
    det = a * c - b * b;
    sDenom = det;

    // Check for (near) parallelism
```

```
    if (det < epsilon) {
        // Arbitrary choice
        sNum = 0;
        tNum = e;
        tDenom = c;
    } else {
        // Find parameter values of closest points
        // on each segment's infinite line. Denominator
        // assumed at this point to be ''det'',
        // which is always positive. We can check
        // value of numerators to see if we're outside
        // the (-inf, inf) x [0, inf) domain.
        sNum = b * e - c * d;
        tNum = a * e - b * d;
    }
    // Check t
    if (tNum < 0) {
        tNum = 0;
        sNum = -d;
        sDenom = a;
    }

    // Parameters of nearest points on restricted domain
    s = sNum / sDenom;
    t = tNum / tDenom;

    // Dot product of vector between points is squared distance
    // between segments
    v = line.base + (s * line.direction) - s1.base + (t * ray.direction);
    return Dot(v,v);
}
```

3. 直线到线段的距离

图 10.33 显示了要求它们之间的距离的一条线段和一条直线，以及其解的有限制定义域。在这种情形中，距离函数的定义域为$[-\infty,\infty]\times[0,1]$（或相反）。定义域由相对的边所包围，对应于$s=0$和$s=1$或$t=0$和$t=1$。如果在线段所在的无限直线上的最接近点的参数小于 0 或大于 1，那么我们就必须在一条边上寻找最接近的点。

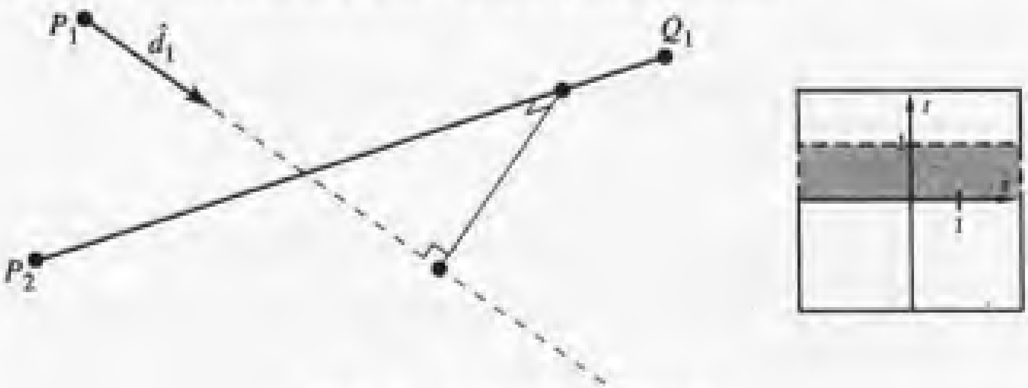


图 10.33　直线与线段之间的距离

伪码为

```
float LineSegmentDistance3D(Line line, Segment s)
{
    u = line.base - seg.base;
    a = Dot(line.direction, line.direction);
    b = Dot(line.direction, seg.direction);
    c = Dot(seg.direction, seg.direction);
    d = Dot(line.direction, u);
    e = Dot(seg.direction, u);
    det = a * c - b * b;
    sDenom = det;

    // Check for (near) parallelism
    if (det < epsilon) {
        // Arbitrary choice
        sNum = 0;
        tNum = e;
        tDenom = c;
    } else {
        // Find parameter values of closest points
        // on each segment's infinite line. Denominator
        // assumed at this point to be ''det'',
        // which is always positive. We can check
        // value of numerators to see if we're outside
        // the [0,1] x [0,1] domain.
        sNum = b * e - c * d;
        tNum = a * e - b * d;
    }

    // Check t
    if (tNum < 0) {
        tNum = 0;
        sNum = -d;
        sDenom = a;
    } else if (tNum > tDenom) {
        tNum = tDenom;
        sNum = -d + b;
        sDenom = a;
    }

    // Parameters of nearest points on restricted domain
    s = sNum / sDenom;
    t = tNum / tDenom;

    // Dot product of vector between points is squared distance
    // between segments
    v = line.base + (s * line.direction) - seg.base + (t * seg.direction);
    return Dot(v,v);
}
```

4. 射线到射线的距离

图 10.34 显示了要求它们之间的距离的两条射线，以及其解的有限制定义域。在这种情形中，距离函数的定义域为$[0,\infty]\times[0,\infty]$。定义域由两条相邻的边所包围，对应于$s=0$和$t=0$。如果在任一射线所在的无限直线上的最接近点的参数小于 0，那么我们就必须在一条边或两条边上寻找最接近的点。

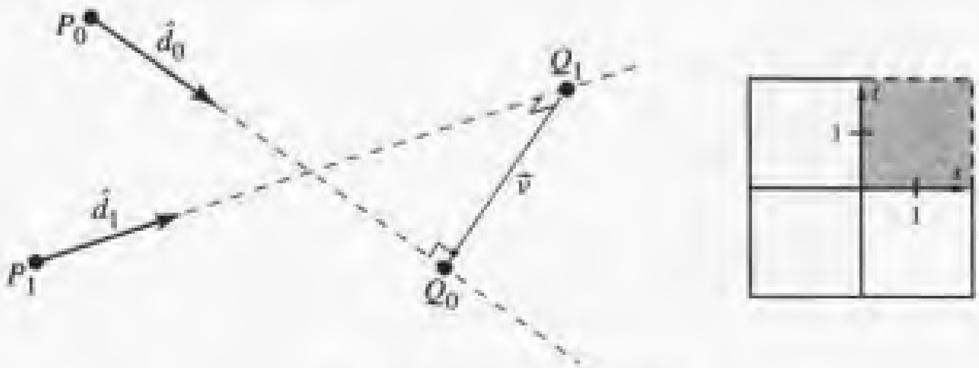


图 10.34 两条射线之间的距离

伪码为

```
float RayRayDistance3D(Ray ray0, Ray ray1)
{
    u = ray0.base - ray1.base;
    a = Dot(ray0.direction, ray0.direction);
    b = Dot(ray0.direction, ray1.direction);
    c = Dot(ray1.direction, ray1.direction);
    d = Dot(ray0.direction, u);
    e = Dot(ray1.direction, u);
    det = a * c - b * b;

    // Check for (near) parallelism
    if (det < epsilon) {
        // Arbitrary choice
        sNum = 0;
        tNum = e;
        tDenom = c;
        sDenom = det;
    } else {
        // Find parameter values of closest points
        // on each segment's infinite line. Denominator
        // assumed at this point to be ''det'',
        // which is always positive. We can check
        // value of numerators to see if we're outside
        // the [0, inf) x [0, inf) domain.
        sNum = b * e - c * d;
        tNum = a * e - b * d;
    }
```

```
    // Check s
    sDenom = det;
    if (sNum < 0) {
        sNum = 0;
        tNum = e;
        tDenom = c;
    }

    // Check t
    if (tNum < 0) {
        tNum = 0;
        if (-d < 0) {
            sNum = 0;
        } else {
            sNum = -d;
            sDenom = a;
        }
    }

    // Parameters of nearest points on restricted domain
    s = sNum / sDenom;
    t = tNum / tDenom;

    // Dot product of vector between points is squared distance
    // between segments
    v = ray0.base + (s * ray0.direction) - ray1.base + (t * ray1.direction);
    return Dot(v,v);
}
```

5. 射线到线段的距离

图 10.35 显示了要求它们之间的距离的一条射线和一条线段，以及其解的有限制定义域。在这种情形中，距离函数的定义域为$[0,\infty]\times[0,1]$（或相反）。定义域由三条边所包围：$s=0$，$s=1$和$t=0$，或者$t=0$，$t=1$和$s=0$。如果在射线所在的无限直线上的最接近点的参数小于 0，或者在线段所在的无限直线上的最接近点的参数小于 0 或大于 1，那么我们就必须在最多两条边上寻找最接近的点。

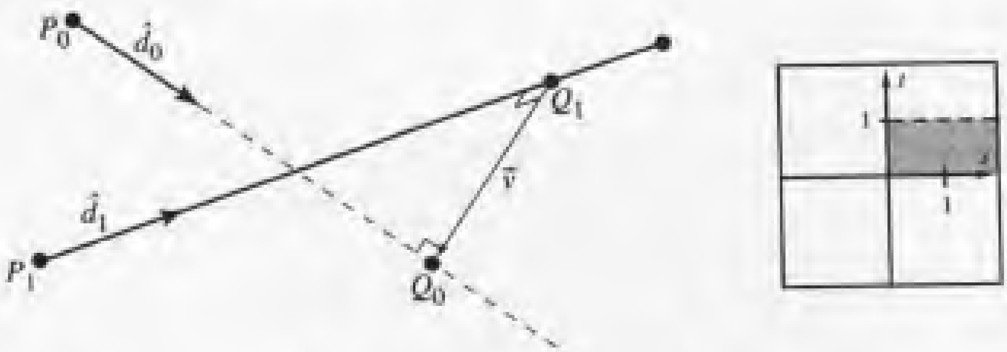


图 10.35　射线和线段之间的距离

伪码为

```
float RaySegmentDistance3D(Ray ray, Segment seg)
{
    u = ray.base - seg.base;
    a = Dot(ray.direction, ray.direction);
    b = Dot(ray.direction, seg.direction);
    c = Dot(seg.direction, seg.direction);
    d = Dot(ray.direction, u);
    e = Dot(seg.direction, u);
    det = a * c - b * b;

    // Check for (near) parallelism
    if (det < epsilon) {
        // Arbitrary choice
        sNum = 0;
        tNum = e;
        tDenom = c;
        sDenom = det;
    } else {
        // Find parameter values of closest points
        // on each segment's infinite line. Denominator
        // assumed at this point to be ''det'',
        // which is always positive. We can check
        // value of numerators to see if we're outside
        // the [0, inf) x [0,1] domain.
        sNum = b * e - c * d;
        tNum = a * e - b * d;
    }

    // Check s
    sDenom = det;
    if (sNum < 0) {
        sNum = 0;
        tNum = e;
        tDenom = c;
    } else {
        tDenom = det;
    }

    // Check t
    if (tNum < 0) {
        tNum = 0;
        if (-d < 0) {
            sNum = 0;
        } else {
            sNum = -d;
            sDenom = a;
        }
    } else if (tNum > tDenom) {
        tNum = tDenom;
```

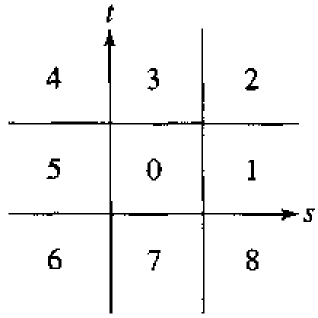


图 10.36　用单位正方形对 st 平面分区

假定 $(\bar{s},\bar{t})$ 在区域 1 内。Q 的阶层曲线就是 st 平面上 Q 为常数所定义的曲线。由于 Q 的图形是一个抛物面，因此其阶层曲线为椭圆。在 $\nabla Q=(0,0)$ 的点上，阶层曲线退化为一个点 $(\bar{s},\bar{t})$。Q 的全局最小值就出现在此处，我们把该最小值称为 $V_{\min}$。由于阶层值 V 从 $V_{\min}$ 开始增大，因此对应的椭圆将不断远离 $(\bar{s},\bar{t})$。存在一个最小的阶层值 V_0，它所对应的椭圆（由 $Q=V_0$ 隐式地定义）仅与单位正方形的边 $s=1$ 接触于一个值 $t=t_0\in[0,1]$。对于阶层值 $V<V_0$，它对应的椭圆与单位正方形不相交。对于阶层值 $V>V_0$，单位正方形的一部分位于它对应的椭圆内。具体而言，这样的椭圆与边的任何交点都必定具有阶层值 $V>V_0$。因此，对于 $t\in[0,1]$ 且 $t\neq t_0$，有 $Q(1,t)>Q(1,t_0)$。点 $(1,t_0)$ 提供了分别位于三维线段上的两个点之间的最小距离。在第一条线段上的是一个端点，在第二条线段上的点是该线段的一个内点。图 10.37 说明了这一论述，其中显示了不同的阶层曲线。

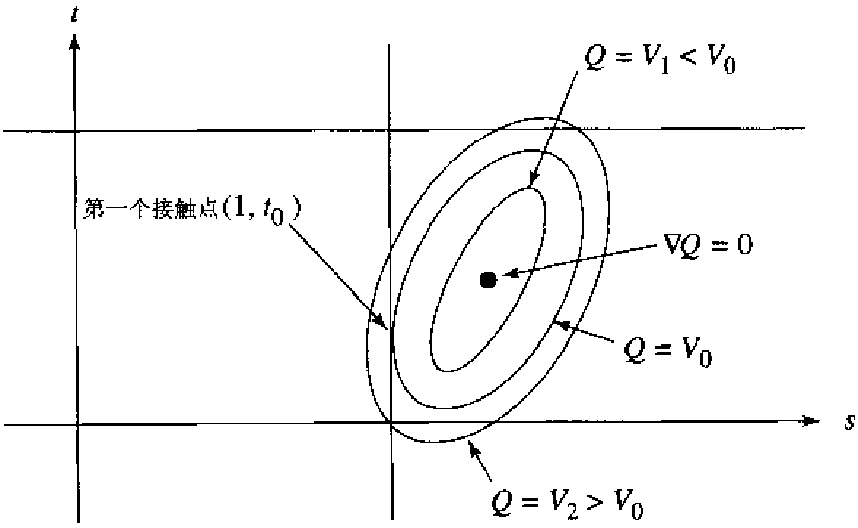


图 10.37　几种不同的阶层曲线 $Q(s,t)=V$

直观地显示具有最小距离的点出现在边界的何处的另一种方法是：将 Q 的图形与平面 $s=1$ 相交。相交得到的曲线是一条抛物线，其图形为 $F(t)=Q(1,t)$（$t\in[0,1]$）。至此，问题已经简化为在一维上求函数 $F(t)$（$t\in[0,1]$）的最小值的问题。$F(t)$ 的最小值或者出现在 $[0,1]$ 的一个内点，在此点上有 $F'(t)=0$，或者出现在端点 $t=0$ 或 $t=1$。图 10.37 显示了最小值出现在内点的情形。在该点上，椭圆与直线 $s=1$ 相切。在位于端点的情形中，椭圆可能与单位正方形的一个角相接触，但并不一定相切。

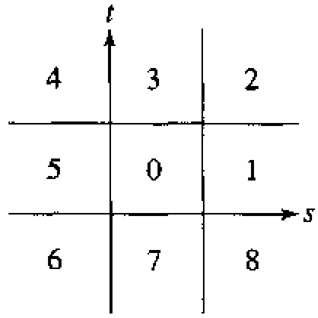


图 10.36 用单位正方形对st平面分区

假定$(\bar{s},\bar{t})$在区域 1 内。Q的阶层曲线就是st平面上Q为常数所定义的曲线。由于Q的图形是一个抛物面，因此其阶层曲线为椭圆。在$\nabla Q=(0,0)$的点上，阶层曲线退化为一个点$(\bar{s},\bar{t})$。Q的全局最小值就出现在此处，我们把该最小值称为$V_{\min}$。由于阶层值V从$V_{\min}$开始增大，因此对应的椭圆将不断远离$(\bar{s},\bar{t})$。存在一个最小的阶层值V_0，它所对应的椭圆（由$Q=V_0$隐式地定义）仅与单位正方形的边$s=1$接触于一个值$t=t_0\in[0,1]$。对于阶层值$V<V_0$，它对应的椭圆与单位正方形不相交。对于阶层值$V>V_0$，单位正方形的一部分位于它对应的椭圆内。具体而言，这样的椭圆与边的任何交点都必定具有阶层值$V>V_0$。因此，对于$t\in[0,1]$且$t\neq t_0$，有$Q(1,t)>Q(1,t_0)$。点$(1,t_0)$提供了分别位于三维线段上的两个点之间的最小距离。在第一条线段上的是一个端点，在第二条线段上的点是该线段的一个内点。图 10.37 说明了这一论述，其中显示了不同的阶层曲线。

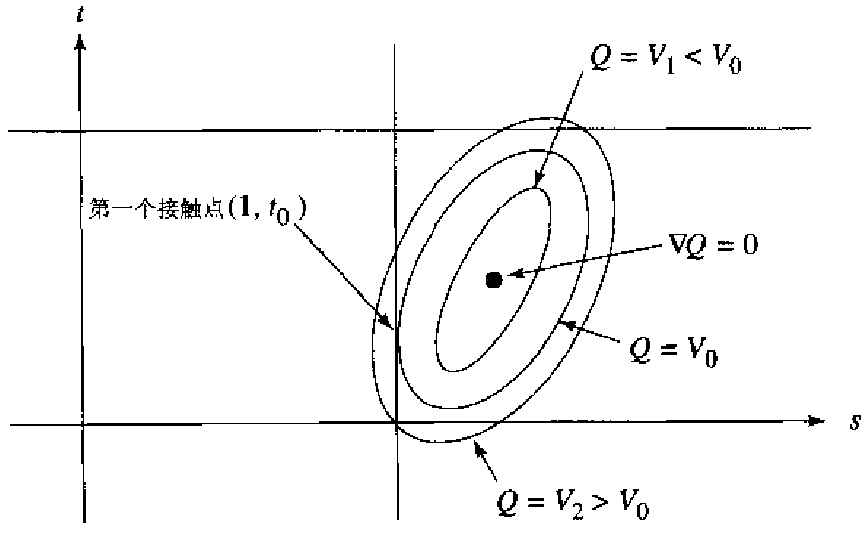


图 10.37 几种不同的阶层曲线 $Q(s,t)=V$

直观地显示具有最小距离的点出现在边界的何处的另一种方法是：将Q的图形与平面$s=1$相交。相交得到的曲线是一条抛物线，其图形为$F(t)=Q(1,t)$（$t\in[0,1]$）。至此，问题已经简化为在一维上求函数$F(t)$（$t\in[0,1]$）的最小值的问题。$F(t)$的最小值或者出现在$[0,1]$的一个内点，在此点上有$F'(t)=0$，或者出现在端点$t=0$或$t=1$。图 10.37 显示了最小值出现在内点的情形。在该点上，椭圆与直线$s=1$相切。在位于端点的情形中，椭圆可能与单位正方形的一个角相接触，但并不一定相切。

为了区分内点和端点的情形，将相同的分区思想应用于一维的情形。区间[0, 1]将实数线划分为三个区间，即$t < 0, t \in [0,1]$和$t > 1$。设$F'(\hat{t}) = 0$。如果$\hat{t} < 0$，那么$F(t)$在$t \in [0,1]$上是递增函数。限制于[0, 1] 内的F的最小值必定出现在$t = 0$处，此时Q在$(s,t) = (1,0)$处取得最小值。如果$\hat{t} > 1$，那么$F(t)$在$t \in [0,1]$上是递减函数。限制于[0, 1]内的F的最小值必定出现在$t = 1$处，此时Q在$(s,t) = (1,1)$处取得最小值。否则，$\hat{t} \in [0,1]$，F在$\hat{t}$处取得最小值，并且Q在$(s,t) = (1,\hat{t})$处取得最小值。

$(\bar{s},\bar{t})$出现在区域 3，5 或 7 内的处理方法与全局最小值出现在区域 1 内的处理方法相似。如果$(\bar{s},\bar{t})$出现在区域 3 内，那么最小值出现在$(s_0,1)$（$s_0 \in [0,1]$）。如果$(\bar{s},\bar{t})$出现在区域 5 内，那么最小值出现在$(0,t_0)$（$t \in [0,1]$）。最后，如果$(\bar{s},\bar{t})$出现在区域 7 内，那么最小值出现在$(s_0,0)$（$s_0 \in [0,1]$）。确定第一个接触点是否位于对应区间的内点或端点的处理方法，与前面所讨论的方法类似。

如果$(\bar{s},\bar{t})$出现在区域 2 内，则Q的阶层曲线与单位正方形的第一个接触点可能出现在边$s = 1$或边$t = 1$上。由于全局最小值出现在区域 2 内，因此位于角(1, 1)处的梯度不可能指向单位正方形内。如果$\nabla Q = (Q_s, Q_t)$，其中Q_s和Q_t是Q的偏微分，那么两个偏微分不可能都为负。可以根据$Q_s(1,1)$和$Q_t(1,1)$的符号来确定选择边$s = 1$或$t = 1$。如果$Q_s(1,1) > 0$，那么最小值必定出现在边$t = 1$上，因为对于$s < 1$且接近于 1，有$Q(s,1) < Q(1,1)$。类似地，如果$Q_t(1,1) > 0$，那么最小值必定出现在边$s = 1$上。确定最小值位于内点还是端点的处理方法，与区域 1 的方法类似。

2. 平行的线段之间的距离

如果$ac - b^2 = 0$，那么Q的梯度在整个st直线上都为零，对所有$t \in \mathbb{R}$，有$s = -(bt + d)/a$。如果任意数对(s,t)在[0, 1]内满足该方程，那么该数对确定了分别位于三维直线上的最接近的两个点。否则，最小值必定出现在单位正方形的边界上。我们并不利用最小值来求解该问题，而是利用线段位于两条平行的直线上这一事实。

假设第一条直线的基点为B_0，方向为$\vec{m}_0$。第一条线段的参数表示为$B_0 + s\vec{m}_0$，其中$s \in [0,1]$。第二条线段可投影到第一条线段上。其端点B_1可表示为

$$B_1 = B_0 + \sigma_0\vec{m}_0 + \vec{u}_0$$

其中$\vec{u}_0$为与$\vec{m}_0$正交的向量。$\vec{m}_0$的系数为

$$\sigma_0 = \frac{\vec{m}_0 \cdot (B_1 - B_0)}{\vec{m}_0 \cdot \vec{m}_0} = -\frac{d}{a}$$

其中a和d为前面定义的$Q(s,t)$的某些系数。另一个端点$B_1 + \vec{m}_1$可表示为

$$B_1 + \vec{m}_1 = B_0 + \sigma_1\vec{m}_0 + \vec{u}_1$$

其中$\vec{u}_1$为与$\vec{m}_0$正交的向量。$\vec{m}_0$的系数为

$$\sigma_1 = \frac{\vec{m}_0 \cdot (\vec{m}_1 + B_1 - B_0)}{\vec{m}_0 \cdot \vec{m}_0} = -\frac{b + d}{a}$$

其中b也是$Q(s,t)$的系数。现在的问题简化为确定对应于[0, 1]的$[\min(\sigma_0,\sigma_1), \max(\sigma_0,\sigma_1)]$的位置。如果两个区间不相连，那么最小距离出现在两条三维线段的端点上。如果两个区间重叠，那么存在许多的点对，在这些点对上具有最小距离。在这种情形中，实现的代码将

返回一对点，一条直线的一个端点和另一条直线的一个内点。

3. 实现

在计算最小距离和对应的最接近的点时，这里给出的实现被设计为最多仅进行一次浮点数除法。而且，除法被推迟到必须执行时才执行。在某些情形中，不需要进行除法。

首先计算在整个算法中都必须用到的数量。具体而言，数值为 $\vec{d} = B_0 - B_1$，$a = \vec{m}_0 \cdot \vec{m}_0$，$b = -\vec{m}_0 \cdot \vec{m}_1$，$c = \vec{m}_1 \cdot \vec{m}_1$，$d = \vec{m}_0 \cdot \vec{d}$，$e = -\vec{m}_1 \cdot \vec{d}$和$f = \vec{d} \cdot \vec{d}$。我们还需要立即确定两条线段是否互相平行。二次函数的分类值为$\delta = ac - b^2$，并且需要先计算出来。代码实际上计算的是$\delta = |ac - b^2|$，因为对于一些接近于平行的直线，某些浮点舍入误差可能导致小的负数。最后，将 δ 与浮点公差值比较。如果较大，那么两条线段是不平行的，并执行处理这种情形的代码。如果较小，那么两条线段是平行的，并执行处理这种情形的代码。

（1）不平行的线段之间的距离

在理论推导上，计算满足$\nabla Q(\bar{s}, \bar{t}) = (0, 0)$的$\bar{s} = (be - cd)/\delta$和$\bar{t} = (bd - ae)/\delta$。然后测试全局最小值的位置，以确定它是否出现在单位正方形[0, 1]内。如果是，那么我们已经确定我们需要计算的最小距离。如果不是，那么必须测试单位正方形的边界。为了推迟与 δ 的除法运算，代码实际上计算的是$\bar{s} = be - cd$和$\bar{t} = bd - ae$，并测试与$[0, \delta]^2$的包含性。如果在集合内，那么执行除法。如果不在，那么测试单位正方形的边界。

确定哪个区域包含$(\bar{s}, \bar{t})$的大致条件为

```
det = a * c - b * b;  s = b * e - c * d;  t = b * d - a * e;
if (s >= 0) {
    if (s <= det) {
        if (t >= 0) {
            if (t <= det) {
                region 0
            } else {
                region 3
            }
        } else {
            region 7
        }
    } else {
        if (t >= 0) {
            if (t <= det) {
                region 1
            } else {
                region 2
            }
        } else {
            region 8
        }
    }
} else {
    if (t >= 0) {
        if (t <= det) {
```

```
                region 5
            } else {
                region 4
            }
        } else {
            region 6
        }
    }
```

处理区域 0 的代码段为

```
invDet = 1 / det;
s *= invDet;
t *= invDet;
```

并且需要进行一次除法。

处理区域 1 的代码段为

```
// F(t) = Q(1, t) = (a + 2 * d + f) + 2 * (b + e) * t + (c) * t^2
// F'(t) = 2 * ((b + e) + c * t)
// F'(T) = 0 when T = -(b + e) / c
s = 1;
tmp = b + e;
if (tmp > 0)  // T < 0, so minimum at t = 0
    t = 0;
else if (-tmp > c)  // T > 1, so minimum at t = 1
    t = 1;
else  // 0 <= T <= 1, so minimum at t = T
    t = -tmp / c;
```

注意，本代码段在运行时，最多进行一次除法运算。处理区域 3，5 和 7 的代码段与此类似。

处理区域 2 的代码段为

```
// Q_s(1, 1) / 2 = a + b + d,  Q_t(1, 1) / 2 = b + c + e
tmp = b + d;
if (-tmp < a)  // Q_s(1, 1) > 0
{
    // F(s) = Q(s, 1) = (c + 2 * e + f) + 2 * (b + d) * s + (a) * s^2
    // F'(s) = 2 * ((b + d) + a * s), F'(S) = 0 when S = -(b + d) / a < 1
    t = 1;
    if (tmp > 0)  // S < 0, so minimum at s = 0
        s = 0;
    else  // 0 <= S < 1, so minimum at s = S
        s = -tmp / a;
} else {
    // Q_s(1,1) <= 0
    s = 1;
    tmp = b + e;
    if (-tmp < c) {
        // Q_t(1,1) > 0
        // F(t) = Q(1,t) = (a + 2 * d + f) + 2 * (b + e) * t + (c) * t^2
```

```
            // F'(t) = 2 * ((b + e) + c * t), F'(T) = 0 when T = -(b + e) / c < 1
            if (tmp > 0)  // T < 0, so minimum at t = 0
                t = 0
            else  // 0 <= T < 1, so minimum at t = T
                t = -tmp / c;
        } else {
            // Q_t(1,1) <= 0, gradient points to region 2, so minimum at t = 1
            t = 1;
        }
    }
```

注意，本代码段在运行时，最多进行一次除法运算。处理区域 4，6 和 8 的代码段与此类似。

（2）平行的线段之间的距离

首先要计算的是$\sigma_0 = -d/a$和$\sigma_1 = -(b+d)/a$的次序。一旦知道了次序，我们就能比较两个 s 区间以确定最小距离。注意$-d/a$对应于$t=0$，且$-(b+d)/a$对应于$t=1$。

```
if (b > 0) {
    // compare intervals [-(b + d) / a, -d / a] to [0, 1]
    if (d >= 0)
        // -d / a <= 0, so minimum is at s = 0, t = 0
    else if (-d <= a)
        // 0 < -d / a <= 1, so minimum is at s = -d / a, t = 0
    else
        // minimum occurs at s = 1, need to determine t (see below)
} else {
    // compare intervals [-d / a, -(b + d) / a] to [0, 1]
    if (-d >= a)
        // 1 <= -d / a, so minimum is at s = 1, t = 0
    else if (d <= 0)
        // 0 <= -d / a < 1, so minimum is at s = -d / a, t = 0
    else
        // minimum occurs at s = 0, need to determine t (see below)
}
```

当$b>0$时，剩下的问题是确定数量$-(b+d)/a$出现在$s=1$的哪一条边上。为了判断，我们首先为对应于$s=1$的$-(bt+d)/a \in [-(b+d)/a, -d/a]$寻找 t 的值。简单地设$-(bt+d)/a = 1$，求得解为$t=-(a+d)/b$。在运行时，当我们得到这种情形时，我们知道$a+d<0$，因此$t>0$。如果 $t \leqslant 1$，那么我们就能直接利用它。但是，如果$t>1$，那么我们将它裁剪为$t=1$。代码段为

```
tmp = a + d;
if (-tmp >= b) t = 1; else t = -tmp / b;
```

再次注意，除法被推迟到需要时才进行。

当$b<0$时，剩下的问题是确定数量$-(b+d)/a$出现在$s=0$的哪一侧。设$-(bt+d)/a=0$，求得解为$t=-d/b$。在运行时，当我们得到此情形时，我们知道$d>0$，因此$t>0$。如果 $t \leqslant 1$，那么我们能直接利用它。但是，如果$t>1$，那么我们将它裁剪为$t=1$。代码段为

```
if (d >= -b) t = 1; else t = -d / b;
```

10.9 线形对象与三角形、矩形、四面体和有向有界箱之间的距离

在本节中，我们将讨论计算线形对象与面形对象（三角形和矩形），以及两种特别的多面体（四面体和有向有界箱）之间的距离。

10.9.1 线形对象到三角形的距离

在本节中，我们考虑计算线形对象和三角形之间的距离问题，如图 10.38 所示。对于这类问题，我们用参数形式来表示线形对象：直线由一个点和一个方向所定义

$$\mathcal{L}(t) = P + t\vec{d}, \quad t \in D_{\mathcal{L}}$$

其中，对于直线，$-\infty \leqslant t \leqslant \infty$；对于射线，$0 \leqslant t \leqslant \infty$；对于线段，$0 \leqslant t \leqslant 1$（其中$P_0$和$P_1$是线段的端点，且$\vec{d} = P_1 - P_0$）。三角形一般表示为三个顶点$V_0$，$V_1$和$V_2$。然而，针对这类问题，我们用参数形式来表示三角形：

$$\mathcal{T}(u, v) = V + u\vec{e}_0 + v\vec{e}_1$$

其中V是三角形的一个顶点，比如V_0，并有$\vec{e}_0 = V_1 - V_0$和$\vec{e}_1 = V_2 - V_0$。那么，三角形上的任意点都可用两个参数u和v来表示，其中$0 \leqslant u$，$v \leqslant 1$且$u + v \leqslant 1$（如图 10.39 所示）。计算线形对象和三角形之间的距离，就意味着必须找到使如下的距离（平方）函数取最小值的t，u和v值

$$Q(u, v, t) = \|\mathcal{T}(u, v) - \mathcal{L}(t)\|^2$$

扩展各项并计算乘积，可得

$$\begin{aligned}
Q(u, v, t) &= \|\mathcal{T}(u, v) - \mathcal{L}(t)\|^2 \\
&= (\vec{e}_0 \cdot \vec{e}_0)u^2 + (\vec{e}_1 \cdot \vec{e}_1)v^2 + (\vec{d} \cdot \vec{d})t^2 \\
&\quad + 2(\vec{e}_0 \cdot \vec{e}_1)uv + 2(-\vec{e}_0 \cdot \vec{d})ut + 2(-\vec{e}_1 \cdot \vec{d})vt \\
&\quad + 2(\vec{e}_0 \cdot (V - P))u + 2(\vec{e}_1 \cdot (V - P))v + 2(-\vec{d} \cdot (V - P))t \\
&\quad + (V - P) \cdot (V - P)
\end{aligned}$$

或更简洁地表示为

$$Q(u, v, t) = a_{00}u^2 + a_{11}v^2 + a_{22}t^2 + 2a_{01}uv + 2a_{02}ut + 2a_{12}vt + 2b_0u + 2b_1v + 2b_2t + c \quad (10.14)$$

其中

$$a_{00} = \vec{e}_0 \cdot \vec{e}_0$$

$$a_{11} = \vec{e}_1 \cdot \vec{e}_1$$

$$a_{22} = \vec{d} \cdot \vec{d}$$

$$a_{01} = \vec{e}_0 \cdot \vec{e}_1$$

$$a_{02} = -\vec{e}_0 \cdot \vec{d}$$

$$a_{12} = -\vec{e}_1 \cdot \vec{d}$$

$$b_0 = \vec{e}_0 \cdot (V - P)$$

$$b_1 = \vec{e}_1 \cdot (V - P)$$

$$b_2 = -\vec{d} \cdot (V - P)$$

$$c = (V - P) \cdot (V - P)$$

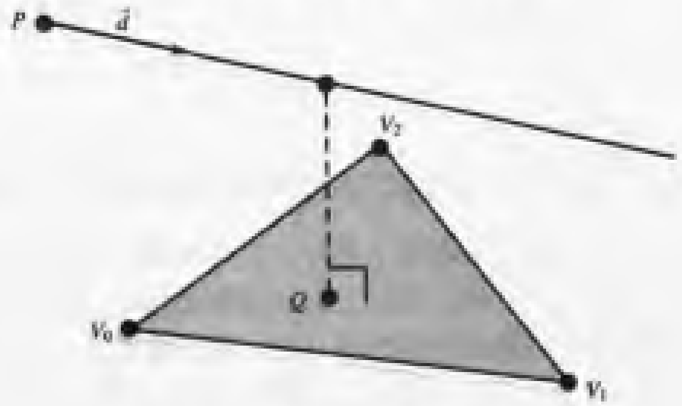


图 10.38 直线与三角形之间的距离

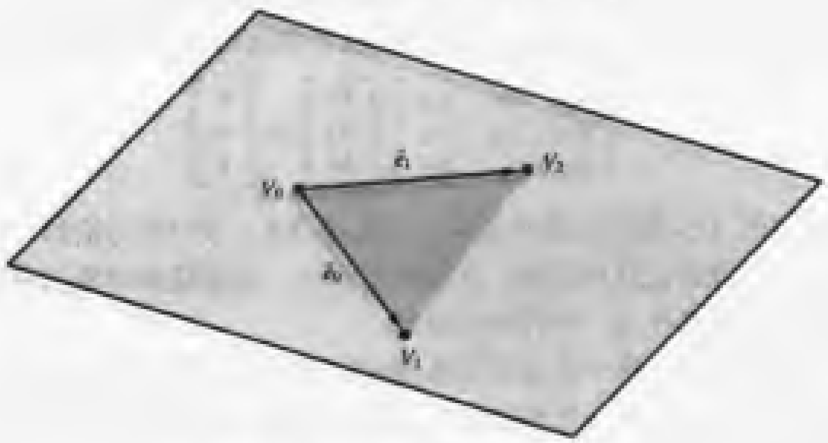


图 10.39 三角形的参数表示形式

由于该问题的解包括三个值(u_0, v_0, t_0)，因此我们可以将其解空间考虑为三个空间维（不要与线形对象和三角形所在的三维空间混淆）。这是模拟显示于图 10.9 中的点一三角形测试的二维解空间域的分区，只是现在我们有一个对应于线形对象的参数 t 作为第三维。对于线形对象，我们有三个可能的域，因此有解空间的三个可能分区分别为：

- 直线：一个无限的三角棱柱
- 线段：一个有限的三角棱柱
- 射线：一个一端无限的三角棱柱

图 10.40 显示了线段的情形。

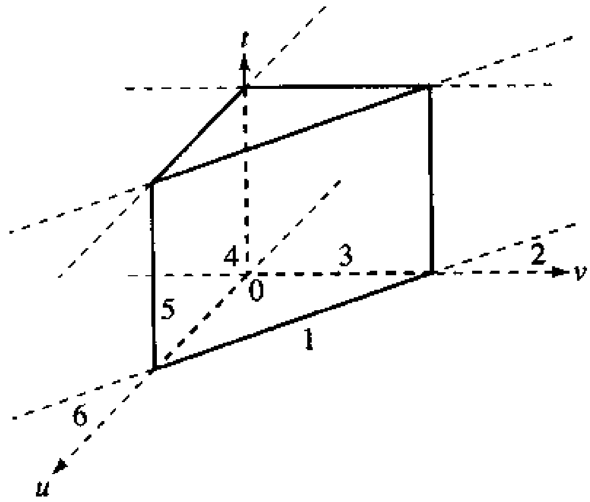


图 10.40　线形对象与三角形之间的距离问题的解空间的可能分区

如果三角形和线形对象的构形满足如下条件，即在包含三角形的平面上最接近于线形对象的点刚好位于三角形内，并且线形对象上与三角形最接近的点刚好位于线段区间内，那么，解$(\bar{u}, \bar{v}, \bar{t})$将位于解域的三角棱柱的区域 0 内，并且线形对象与三角形之间的距离为$\|\mathcal{T}(\bar{u}, \bar{v}, \bar{t}) - \mathcal{L}(t)\|$。否则，$\nabla Q = (\bar{u}, \bar{v}, \bar{t})$的最小值将在一个分隔区域的面上（并且，它在面上所处的位置可能属于一条边或者是区域边界的一个顶点）。

算法的第一步是寻找使方程（10.14）中的距离平方函数取最小值的u，v和t。严格地说，我们必须找到使$\nabla Q = 0$的位置，这可以表示为寻找如下方程系统的解：

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} u \\ v \\ t \end{bmatrix}$$

这可以通过求其中的 3×3 矩阵的逆来实现。正如在 2.5.4 节中所讨论的，一个矩阵必须具有非零的行列式，才能保证是可逆的。在这种情形中，如果线形对象与平面平行，那么行列式为零，我们可以将其作为一种特例来处理。

对于每一种线形对象，第一步是计算系统的行列式，并且如果它为非零值，那么计算其解，以找到解所在的区域。一旦确定了该区域，就进行针对该区域的计算，以求得表示最小值的边界上的点$(\bar{u}, \bar{v}, \bar{t})$。

1. 直线与三角形之间的距离

对于直线与三角形之间的距离，解域将域划分为 6 个区域。确定区域的伪码如下：

```
float LineTriangleDistanceSquared(Line line, Triangle triangle)
{
    e0 = triangle.v[1] - triangle.v[0];
    e1 = triangle.v[2] - triangle.v[0];
    a00 =  Dot(e0, e0);
    a01 =  Dot(e0, e1);
    a02 = -Dot(e0, line.direction);
    a11 =  Dot(e1, e1);
```

```
a12 = -Dot(e1, line.direction);
a22 =  Dot(line.direction, line.direction);
diff = triangle.v[0] - line.base;
b0 =  Dot(e0, diff);
b1 =  Dot(e1, diff);
b2 = -Dot(line.direction, diff);
c = Dot(diff, diff);

//  Cofactors to be used for determinant
//  and inversion of matrix A
cof00 = a11 * a22 - a12 * a12;
cof01 = a02 * a12 - a01 * a22;
cof02 = a01 * a12 - a02 * a11;
det = a00 * cof00 - a01 * cof01 + a02 * cof02;

//  Invert determinant and b if det is negative --
//  Avoids having to deal with special cases later.
if (det < 0) {
    det = -det;
    b0 = -b0;
    b1 = -b1;
    b2 = -b2;
}

//  Check if determinant is (nearly) 0
if (det < epsilon) {
    //  Treat line and triangle as parallel. Compute
    //  closest points by computing distance from
    //  line to each triangle edge and taking minimum.
    Segment seg0 = {triangle.v[0], triangle.v[1] };
    dist0 = LineLineSegDistanceSquared(line, seg0);

    Segment seg1 = {triangle.v[0], triangle.v[1] };
    dist1 = LineLineSegDistanceSquared(line, seg1);

    Segment seg2 = {triangle.v[1], triangle.v[2] };
    dist2 = LineLineSegDistanceSquared(line, seg2);

    distance = MIN(dist0, MIN(dist1, dist2));

    return distance;
} else {
    //  Determine the region in which solution lies by
    //  computing u and v and checking their signs
    cof11 = a00 * a22 - a02 * a02;
    cof12 = a02 * a01 - a00 * a12;
    u = -(cof00 * b0 + cof01 * b1 + cof02 * b2);
    v = -(cof01 * b0 + cof11 * b1 + cof12 * b2);

    if (u + v <= det) {
        if (u < 0) {
```

```
                    if (v < 0) {
                        region 4
                    } else {
                        region 3
                    }
                } else if (v < 0) {
                    region 5
                } else {
                    region 0
                }
            } else {
                if (u < 0) {
                    region 2
                } else if (v < 0) {
                    region 6
                } else {
                    region 1
                }
            }
        }
    }
```

处理区域0的伪码为

```
invDet = 1 / det;
u = u * invDet;
v = v * invDet;

cof22 = a00 * a11 - a01 * a01;
r = -(cof02 * b0 + cof12 * b1 + cof22 * b2) * invDet;
```

处理其他区域的代码更复杂一些。考虑处理区域3的代码，此时$\bar{u}=0$。如果我们将它代入方程（10.14），即原来的距离平方公式，就去掉了与u有关的项，并且有效地得到了一个较低维的二次方程：

$$Q_1(v,t)=a_{11}v^2+a_{22}t^2+2a_{12}vt+2b_1v+2b_2t+c,\quad v,t\in[0,1]\times(-\infty,\infty)$$

该区域包含一个无限带，该带由位于(v,t)平面内的$v=0$和$v=1$，以及两个半平面所包围，在图10.41中，显示为“从t轴往下看”。计算∇Q_1的解$(\bar{v},\bar{t})$，v必须位于无限带或在其中的一个半平面内。如果它位于无限带内（$0\leqslant\bar{v}\leqslant1$），那么$Q$的解为$(0,\bar{v},\bar{t})$。否则，它位于其中的一个半平面内，并且最小值将位于无限带和半平面相交的直线上（$v=0$或$v=1$）。如果$v=0$，那么从方程（10.14）中去掉u和v的项后得到的二次方程为

$$Q_2(t)=a_{22}t^2+2b_2t+c,\quad t\in(-\infty,\infty)$$

当满足如下条件时，出现解

$$\frac{dQ_2}{dt}=0$$

因此

$$t=\frac{-b_2}{a_{22}}$$

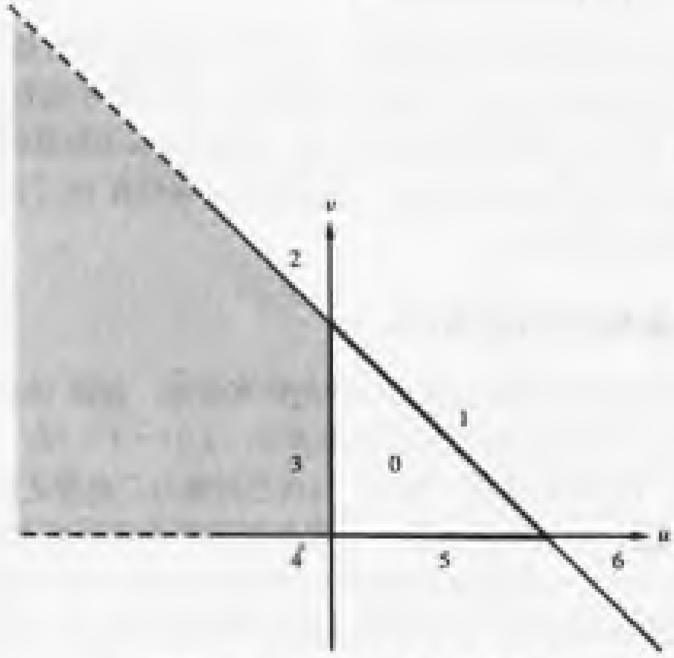


图 10.41　区域 3 的边界带和边界平面

如果$\bar{v}>1$，那么需要求最小值的二次函数为

$$Q_3=a_{22}t^2+2(a_{12}+b_2)t+(a_{11}+2b_1+c)$$

因此

$$t=-\frac{a_{12}+b_2}{a_{22}}.$$

处理这种情况的伪码为

```
u = 0;
v = a12 * b2 - a22 * b1;
if (t >= 0) {
    if (t <= det) {
        invDet = 1 / cof00;
        v *= invDet;
        t = (a12 * b1 - a22 * b2) * invDet;
    } else {
        v = 1;
        t = -(b2 + a12) / a22;
    }
} else {
    v = 0;
    t = -b2 / a22;
}
```

处理区域 1 和 5 的相关代码和分析与区域 3 的情况相似。处理区域 2，4 和 6 的相关代码和分析相对简单，因为我们仅需测试两个半平面：分别对于每一个区域，我们有

$(u=0, v=1)$，$(u=0, v=0)$和$(u=1, v=0)$。

2．射线和线段与三角形之间的距离

计算射线与三角形之间的距离和线段与三角形之间的距离的基本方法完全类似于计算直线与三角形之间的距离的方法。然而，对于前者，在域中不是有 6 个区域，而是有 12 个区域（其中 6 个与直线/三角形的情形相同，由于射线将其无限直线划分为两个区域，即 $t<0$和$t\geqslant 0$，所以区域数加倍。对于后面一种的情形，我们有 18 个区域，即 3 组相同的 6 个区域，即$t<0$，$0\leqslant t\leqslant 1$ 和$t>1$。

10.9.2　线形对象到矩形的距离

我们在本节讨论计算线形对象与矩形之间的距离问题，如图 10.42 所示。线形对象用常用的表示法来表示，即用基点和方向向量来表示：$\mathcal{L}(t)=P+t\vec{d}$。一般地，矩形可认为是由 4 个顶点$V_0$，$V_1$，$V_2$和$V_3$定义的。然而，与线形对象与三角形之间的距离的讨论一样，我们利用另一种表示法，即认为矩形由一个顶点和连接于该顶点上的两条边所定义。我们任意选取V_0为原点，定义一个矩形$\{V, \vec{e}_0, \vec{e}_1\}$，其中$V=V_0$，$\vec{e}_0=V_1-V_0$和$\vec{e}_1=V_3-V_0$，如图 10.13 所示。这样我们得到一个参数形式的矩形表示$\mathcal{R}(u,v)=V+u\vec{e}_0+v\vec{e}_1$，其中 $0\leqslant u$，$v\leqslant 1$。

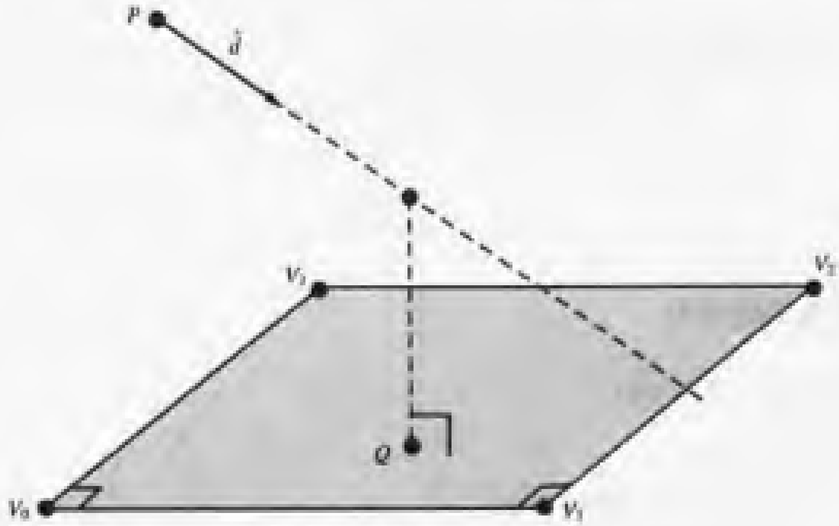


图 10.42　直线与矩形之间的距离

这样，我们的平方距离函数为

$$Q(u,v,t)=\|\mathcal{R}(u,v)-\mathcal{L}(t)\|.$$

代入线形对象和矩形的公式，可得

$$\begin{aligned}Q(u,v,t)=&(\vec{e}_0\cdot\vec{e}_0)u^2+(\vec{e}_1\cdot\vec{e}_1)v^2+(\vec{d}\cdot\vec{d})t^2\\&+(\vec{e}_0\cdot\vec{e}_1)uv+(-\vec{e}_0\cdot\vec{d})ut+(-\vec{e}_1\cdot\vec{d})vt\\&+2(\vec{e}_0\cdot(V-P))u+2(\vec{e}_1\cdot(V-P))v+2((V-P)\cdot(V-P))t+c\end{aligned}$$

或者更简洁一些，

$$Q(u,v,t) = a_{00}u^2 + a_{11}v^2 + a_{22}t^2 + a_{01}uv + a_{02}ut + a_{12}vt + 2b_0u + 2b_1v + 2b_2t + c \quad (10.15)$$

其中

$$a_{00} = \vec{e}_0 \cdot \vec{e}_0$$
$$a_{01} = \vec{e}_0 \cdot \vec{e}_1$$
$$a_{11} = \vec{e}_1 \cdot \vec{e}_1$$
$$a_{12} = -\vec{e}_0 \cdot \vec{d}$$
$$a_{02} = -\vec{e}_0 \cdot \vec{d}$$
$$a_{12} = -\vec{e}_1 \cdot \vec{d}$$
$$a_{22} = \vec{d} \cdot \vec{d}$$
$$b_0 = \vec{e}_0 \cdot (V - P)$$
$$b_1 = \vec{e}_1 \cdot (V - P)$$
$$b_2 = (V - P) \cdot (V - P)$$

Q的定义域为$\mathbb{R}^3$，并用类似于图 10.40 的方法来进行分区，只是分别针对直线、射线和线段的情形分别定义无限的正方形列、半无限的正方形列和立方体。针对线段的分区情形如图 10.43 所示。如果矩形和线形对象的构形满足如下条件，即在包含矩形的平面上，最接近于线形对象的点刚好位于矩形内，并且线段上最接近于矩形的点位于线段的区域内，那么解$(\bar{u},\bar{v},\bar{t})$将位于解域立方体的区域 0 内；线形对象与矩形之间的距离为$\|\mathcal{T}(\bar{u},\bar{v},\bar{t}) - \mathcal{L}(t)\|$。否则，$\nabla Q = (\bar{u},\bar{v},\bar{t})$的最小值将在一个分隔区域的面上（并且，它在面上所处的位置可能属于一条边或者是区域边界的一个顶点）。

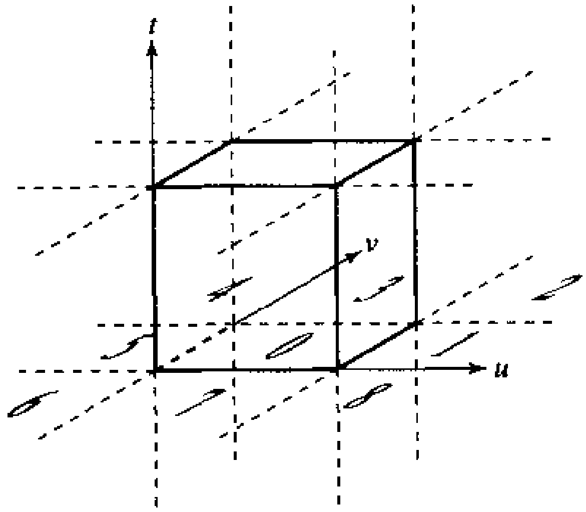


图 10.43 线段与矩形之间距离的定义域的可能分区

算法的第一步是寻找使方程（10.15）中的距离平方函数取最小值的u，v和t。严格地说，我们必须找到使$\nabla Q = 0$的位置，这可表示为寻找如下方程系统的解：

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} u \\ v \\ t \end{bmatrix}$$

这可通过求其中的 3×3 矩阵的逆来实现。正如在 2.5.4 节中所讨论的，一个矩阵必须具有非零的行列式，才能保证是可逆的。在这种情形中，如果线形对象与平面平行，那么行列式为零，我们可以将其作为一种特例来处理。

对于每一种线形对象，第一步是计算系统的行列式，并且如果它为非零，那么计算其解，以找到解所在的区域。一旦确定了该区域，就进行针对该区域的计算，以求得表示最小值的边界上的点$(\bar{u}, \bar{v}, \bar{t})$。

1．直线与矩形之间的距离

对于直线与矩形之间的距离，解域划分为一个包含 9 个区域的正方形列。确定无约束条件的解所在的区域的伪码如下：

```
float LineRectangleDistanceSquared(Line line, Rectangle rectangle)
{
    // Convert to parametric representation
    // Assumes rectangle vertices are ordered counterclockwise
    V = rectangle.V[0];
    e0 = rectangle.V[1] - V;
    e1 = rectangle.V[3] - V;

    a00 =  Dot(e0, e0);
    a01 =  Dot(e0, e1);
    a02 = -Dot(e0, line.direction);
    a11 =  Dot(e1, e1);
    a12 = -Dot(e1, line.direction);
    a22 =  Dot(line.direction, line.direction);
    diff = V - line.base;
    b0 = Dot(e0, diff);
    b1 = Dot(e1, diff);
    b2 = Dot(line.direction, diff);
    c = Dot(diff, diff);

    // Cofactors to be used for determinant
    // and inversion of matrix A
    cof00 = a11 * a22 - a12 * a12;
    cof01 = a02 * a12 - a01 * a22;
    cof02 = a01 * a12 - a02 * a11;
    det = a00 * cof00 + a01 * cof01 + a02 * cof02;

    // Flip determinant and b, to avoid
    // special cases later
    if (det < 0) {
        det = -det;
        b0 = -b0;
        b1 = -b1;
        b2 = -b2;
```

```
}

// Check for (near) parallelism
if (det < epsilon) {
    // Line and rectangle are parallel.
    // Find the closest points by computing
    // the distance between the line to
    // the segment defining each edge and
    // taking the minimum.
    Segment seg0 = { rectangle[0], rectangle[1] };
    float dist0 = LineLineSegmentDistanceSquared(line, seg0);

    Segment seg1 = { rectangle[1], rectangle[2] };
    float dist1 = LineLineSegmentDistanceSquared(line, seg1);

    Segment seg2 = { rectangle[2], rectangle[3] };
    float dist2 = LineLineSegmentDistanceSquared(line, seg2);

    Segment seg3 = { rectangle[3], rectangle[0] };
    float dist3 = LineLineSegmentDistanceSquared(line, seg3);

    return (MIN(seg0, MIN(seg1, MIN(seg2, seg3))));
}

// Compute u, v
cof11 = a00 * a22 - a02 * a02;
cof12 = a02 * a01 - a00 * a12;
u = -(cof00 * b0 + cof01 * b1 + cof02 * b2);
v = -(cof01 * b0 + cof11 * b1 + cof12 * b2);

if (s < 0) {
    if (t < 0) {
        region 6;
    } else if (t <= det) {
        region 5;
    } else {
        region 4;
    }
} else if (s <= det) {
    if (t < 0) {
        region 7;
    } else if (t <= det) {
        region 0;
    } else {
        region 3;
    }
} else {
    if (t < 0) {
        region 8;
```

```
        } else if (t <= det) {
            region 1;
        } else {
            region 2;
        }
    }
```

处理不同区域的代码实现与处理直线与三角形之间的距离的代码实现方式完全相同。

2. 射线和线段与矩形之间的距离

计算射线与矩形之间的距离和线段与矩形之间的距离的基本方法，完全类似于计算直线与矩形之间的距离的方法。然而，对于前者，在域中不是有 9 个区域，而是有 18 个区域（其中 9 个与直线/矩形的情形相同，由于射线将其无限直线划分为两个区域，即 $t<0$ 和 $t\geqslant 0$，所以区域数加倍。对于后者的情形，我们有 27 个区域，即 3 组相同的 9 个区域，即 $t<0$，$0\leqslant t\leqslant 1$ 和 $t>1$。

10.9.3　线形对象到四面体的距离

我们在本节将讨论计算线形对象与四面体之间的距离问题，如图 10.44 所示。设 V_i（$0\leqslant i\leqslant 3$）为四面体的顶点。线形对象为 $P+t\hat{d}$，其中 $\hat{d}$ 为单位长度方向向量，$t\in\mathbb{R}$（直线），$t\geqslant 0$（射线）或 $t\in[0,T]$（线段）。对于不是单位长度的 $\hat{d}$，需要对构造做一点修改。四面体可参数化表示为 $V_0+s_1\vec{e}_1+s_2\vec{e}_2+s_3\vec{e}_3$，其中 $\vec{e}_i=V_i-V_0$，$s_i\geqslant 0$ 且 $s_1+s_2+s_3\leqslant 1$。

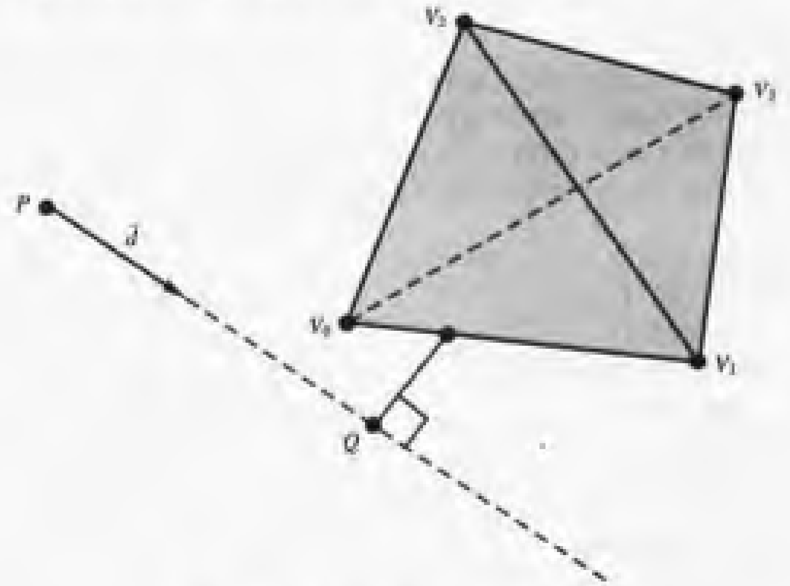


图 10.44　直线与四面体之间的距离

1. 距离

通过减去 P 来实现平移四面体和直线。对于所有的 i，四面体的顶点现在成为 $V_i=U_i-P$。直线成为 $s\hat{d}$。将它们投影到包含原点 O 且法线为 $\hat{d}$ 的平面上。图 10.45（a）显示了这种投影（为了清晰，显示在二维空间中）。投影得到的直线是一个点 O。投影所得的四面体的顶点为 $W_i=(I-\hat{d}\hat{d}^{\mathrm{T}})V_i$。投影所得的实心四面体的边界是一个凸多边形，是一

个三角形或者四边形。如果凸多边形包含O，那么从直线到四面体的距离为零。否则，从直线到四面体的距离就是从O到凸多边形的距离。投影值位于一个三维空间的平面内，并且可以用标准的方法来消去对应于$\hat{d}$的最大绝对值分量的坐标，以将其投影到二维空间上，如图 10.45（b）所示。可以在二维空间上计算点与多边形之间的距离。为了适应三维到二维的投影，必须校正该值。例如，如果$\hat{d}=(d_0,d_1,d_2)$，其中$|d_2|=\max_i\{|d_i|\}$且r是计算得到的二维距离，那么三维距离就是r/d_2。

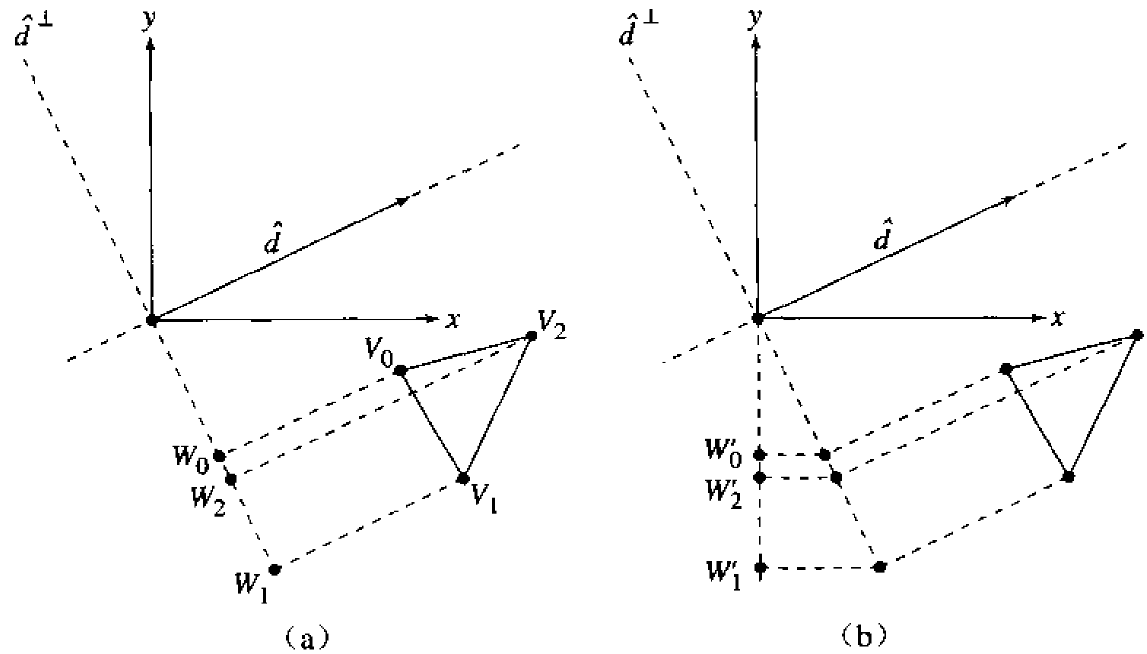


图 10.45　四面体先投影到（a）垂直于$\hat{d}$的平面上，然后投影到（b）二维空间

2. 最接近的点

四面体上最接近于直线的点集在许多情况下仅仅包含单个的点。在另外的一些情形中，该集合可能包含一条线段上的点。例如，考虑顶点为（0，0，0），（1，0，0），（0，1，0）和（0，0，1）的四面体。当$t\in[0,1/2]$时，直线$(1/4,1/4,0)+t(0,0,1)$与该四面体相交。因此，对应的点与四面体的距离为零单位长度。直线$(-1,-1,1/2)+t(0,0,1)$与四面体的距离为$\sqrt{2}$单位。直线上最接近的点由$t\in[0,1/2]$产生，而四面体上最接近的点为（0，0，t）（对于相同的t值区间）。直线$(1/2,-1/2,0)+t(0,0,1)$与四面体的距离为 1/2 单位。直线上最接近的点由$t\in[0,1/2]$产生，而四面体上最接近的点为（1/2，0，t）（对于相同的t值区间）。

【情形 1】设O严格位于凸多边形内。在这种情形中，直线与四面体相交于一个区间内的点。设$\mathrm{E}=[\vec{e}_1\;\vec{e}_2\;\vec{e}_3]$为各列为指定的四面体的边向量的矩阵。设$\vec{s}$为分量为$s_i$参数的 3×1 向量。相交的线段为$t\hat{d}+P=\mathrm{E}\vec{s}+V_0$（$t\in[t_{\min},t_{\max}]$）。现在要解决的问题是计算$t$区间。四面体的边向量是线性无关的，因此E是可逆的。用其逆矩阵乘以向量方程的两边，并求解四面体的参数，可得

$$\vec{s}=\mathrm{E}^{-1}\left(t\hat{d}+P-V_0\right)=\vec{a}t+\vec{b}$$

其中 $\vec{a}=(a_1,a_2,a_3)=\mathrm{E}^{-1}\hat{d}$ 且 $\vec{b}=(b_1,b_2,b_3)=\mathrm{E}^{-1}(P-V_0)$。参数 $\vec{s}$ 必须满足四面体的不等式限制条件。因此，参数t由如下的 4 个不等式来限制：

$$a_1t+b_1\geqslant 0,\ a_2t+b_2\geqslant 0,\ a_3t+b_3\geqslant 0,\ (a_1+a_2+a_3)t+(b_1+b_2+b_3)\leqslant 1$$

每一个不等式都定义一个形式为$[\bar{t},\infty)$或$(-\infty,\bar{t}]$的半无限区间。在这种特殊情形中，我们知道相交的 4 个区间必须为非空的，且形式为$[t_{\min},t_{\max}]$。

可以避免计算E^{-1}所需的除法。我们假设四面体是有向的，因此$\det(\mathrm{E})>0$。乘以伴随矩阵，$\mathrm{E}^{\mathrm{adj}}$可得

$$\det(\mathrm{E})\vec{s}=\mathrm{E}^{\mathrm{adj}}\left(t\hat{d}+P-V_0\right)=\vec{\alpha}t+\vec{\beta}$$

4 个关于t 的不等式与上面的不等式的形式相同，但是a_i表示$\vec{\alpha}$的分量，b_i 表示$\vec{\beta}$的分量，并且最后的一个不等式是与$\det(\mathrm{E})$比较，而不是与 1 比较。■

【情形 2】设O位于凸多面体的边界上或者位于多边形外。设C为最接近于O的多边形点（在三维空间中）。直线$t\hat{d}+C$与顶点为U_i的四面体相交于一个点或者一个点区间。可以利用案例 1 中的方法，但是当使用浮点算术时，我们应该慎重地构建区间。如果交点是一个点，那么在理论上有$t_{\min}=t_{\max}$，但是在计算所得的数值上，你可能得到一个空区间。正确地处理这种情况并不困难。注意，由于在案例 1 中可以选择$C=O$，因此可用相同的代码来处理案例 1 和案例 2。■

3. 射线与四面体之间的距离

使用直线－四面体算法来计算参数为$I=[t_{\min},t_{\max}]$（可能有$t_{\min}=t_{\max}$）的最接近的直线点。定义$J=I\cap[0,\infty)$。如果$J\neq\emptyset$，那么射线－四面体之间的距离与直线－四面体之间的距离相同。最接近的射线点由J确定。如果$J=\emptyset$，那么射线的端点P最接近于四面体。

4. 线段与四面体之间的距离

使用直线－四面体算法来计算参数为$[t_{\min},t_{\max}]$（可能有$t_{\min}=t_{\max}$）的最接近的直线点。定义$J=I\cap[0,T]$。如果$J\neq\emptyset$，那么线段－四面体之间的距离与直线－四面体之间的距离相同。最接近的射线点由J确定。如果$J=\emptyset$，那么最接近的线段点为：当$t_{\max}<0$时，为P；当$t_{\min}>T$时，为$P+T\hat{d}$。

10.9.4 线形对象到有向有界箱的距离

本节我们将讨论计算线形对象与有向有界箱之间的距离问题。线形对象定义为一个基点和一个方向向量：

$$\mathcal{L}(t)=P+t\vec{d}$$

有向有界箱定义为一个中心C，三个正交方向向量$\hat{u}$，$\hat{v}$和$\hat{w}$，以及三个宽度的一半h_u，h_v和h_w，如图 10.46 所示。

最简单的方法就是计算直线与 6 个面之间的距离平方值，并求它们中的最小值。然而，这种方法是非常费时的，因为各个面的方向具有任意性，也不能利用每一个面与其他的面不是互相平行就是互相垂直的性质。

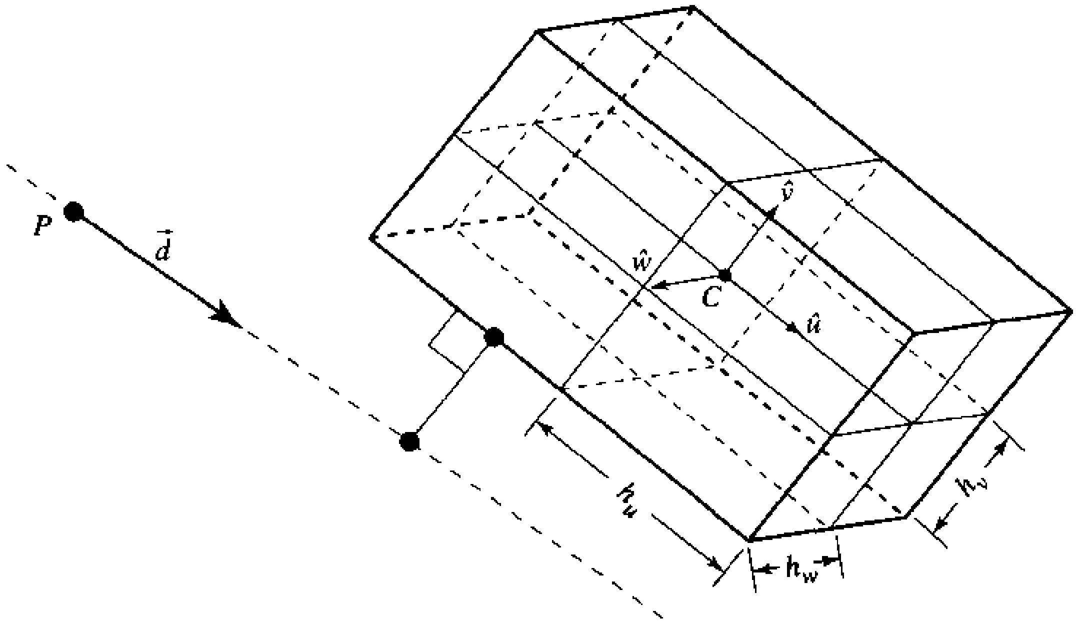


图 10.46　直线与有向有界箱之间的距离

我们在此采用的方法类似于 10.4.2 节中介绍的计算点与有向有界箱之间的距离的方法。

有向有界箱的中心C和三个正交方向向量$\hat{u}$，$\hat{v}$和$\hat{w}$定义了一个坐标系。我们注意到，相对于该坐标系，有向有界箱是一个中心位于原点的轴对齐箱，它的各条边分别与x轴，y轴或z轴平行。如果我们将直线$\{P,\vec{d}\}$转换到该坐标系中，那么我们就能利用有向有界箱的性质，即轴对齐、中心位于原点，并计算最接近的点及与该点的距离，这样效率更高。

有向有界箱的中心C和三个正交方向向量$\hat{u}$，$\hat{v}$和$\hat{w}$定义了一个点P在该坐标系中的坐标，可以表示为

$$P' = [\,(P-C)\cdot\hat{u}\quad (P-C)\cdot\hat{v}\quad (P-C)\cdot\hat{w}\,]$$

即只需简单地将P和C构成的向量投影到有向有界箱的每一个基底向量上。注意，在功能上，这等价于建立一个从全局空间到有向有界箱的局部的变换T与P的乘积，但效率更高。其变换T的变换矩阵为

$$\mathbf{T} = \begin{bmatrix} \hat{u}^{\mathrm{T}} & \hat{v}^{\mathrm{T}} & \hat{w}^{\mathrm{T}} & \vec{0}^{\mathrm{T}} \\ & -C & & 1 \end{bmatrix}$$

也可以对$\vec{d}$（直线的方向向量）进行相同的处理。

一旦进入有向有界箱的坐标系，我们就能计算如下的三种数值：

- 有向有界箱上最接近于直线的点的坐标。
- 直线上最接近于有向有界箱的点的参数。
- 有向有界箱与直线之间的距离的平方。

注意，如果直线与有向有界箱相交，它们之间的距离当然为零，其他的两项信息也就没有意义了。

有向有界箱上最接近于直线的点的坐标可在原点为［0，0，0］，基底向量为正交的坐

标系中找到。因此，我们可以将这些坐标看成有向有界箱上最接近的点在未变换的坐标系中的的参数值。我们只需简单地计算下式：

$$Q = C + Q'_x\hat{u} + Q'_y\hat{v} + Q'_z\hat{w}$$

当然，为了找到在未变换的坐标系内的确定点（如果它是要计算的），变换后的直线上的点的参数可简单地应用于未变换的直线。图 10.47 显示了这种处理的示意图（为清晰起见，显示在二维空间中）。

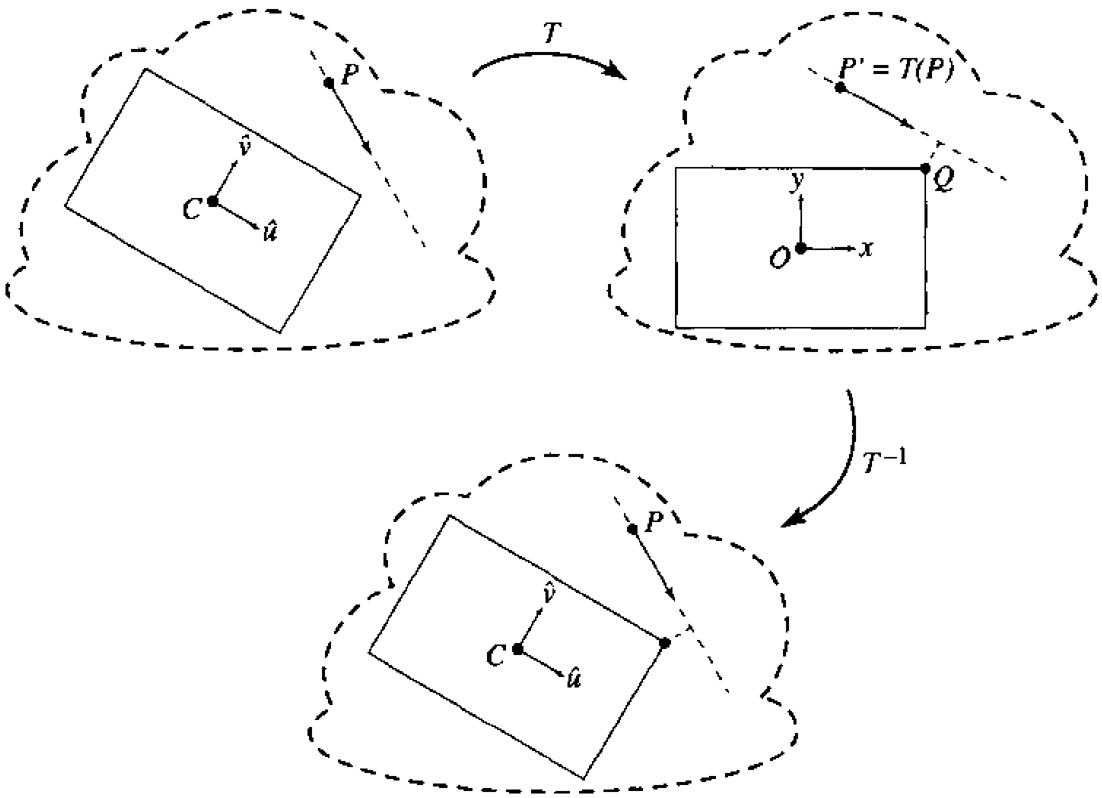


图 10.47　直线与 OBB 之间的距离的求解算法示意图

一旦直线被变换到有向有界箱的本地坐标系内，我们就能利用有向有界箱的各个面分别平行于坐标系的各基底向量这一事实。自然，当“目标”平行于基底向量时，计算相交和距离的效率更高，然而，我们通过将它们之间的构形分解为不同的情形，对不同的情形使用仅在需要时才计算最小距离的算法，可以更进一步地利用该性质。

通过观察直线$\mathcal{L}$的（变换后的）方向向量$\vec{d}'$，可以对构形进行分类。特别地，我们分析有几个$\vec{d}'$的分量为零，其数目可能是没有、一个、两个或三个。为了不增加复杂性，我们分别讨论这些情形。

对于没有、一个或两个分量的情形，我们在实现中使用一点“技巧”。我们考虑$\vec{d}'$的分量的值：如果一个分量是负数，那么我们反转其符号，并反转对应的P'的分量的符号。然后，我们进行其余的算法，并反转对应的有向有界箱的上最接近的点的分量（因为有向有界箱在三个维上都关于原点对称，因此可以这样处理）。这将保证$\vec{d}'$的所有分量都是非负的，从而减少我们必须考虑的不同情形的数量。

1. 三个零分量

在这种情形中，直线就是一个点，因此问题简化为计算点到一个轴对齐的有向有界箱的距离，在 10.4.2 节中可以找到求解方法。

2. 两个零分量

在这种情形中，变换后的直线垂直于两个基底向量并平行于另一个基底向量。图 10.48 显示了垂直于有向有界箱的 y 和 z 轴的一条直线，它可能在有向有界箱的上面、里面或下面。

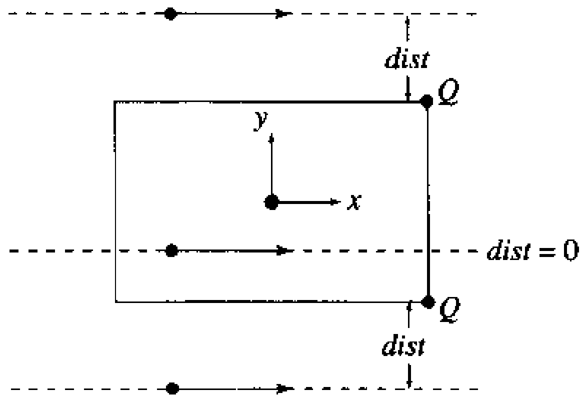


图 10.48　两个零分量的情形

在这种情形中，距离平方就是在 y 和 z 上的距离平方之和：

$$d_y^2 = \begin{cases} P_y'^2 + \text{extent}_y & P_y' < -\text{extent}_y \\ P_y'^2 - \text{extent}_y & P_y' > \text{extent}_y \\ 0 & \text{否则} \end{cases}$$

$$d_z^2 = \begin{cases} P_z'^2 + \text{extent}_z & P_z' < -\text{extent}_z \\ P_z'^2 - \text{extent}_z & P_z' > \text{extent}_z \\ 0 & \text{否则} \end{cases}$$

注意，如果直线与有向有界箱相交，距离就为零。

直线上最接近的点可以（任意但一致地）选取为对应于有向有界箱的"正" YZ 面的点：

$$t = \frac{\text{extent}_x - P_x'}{\vec{d}'}$$

有向有界箱上最接近的点的可以（任意但一致地）选取在有向有界箱的"正" YZ 面上；如果直线与有向有界箱相交且各沿一条边的方向，那么该点就在面内；否则就位于一个角上。

伪码为

```
real CaseTwoZeroComponents_YZ(Line line, OBB box, real t)
{
    real distanceSquared = 0;

    // Parameter of closest point on the line
    t = (box.extents.x - line.origin.x) / line.direction.x;
```

```
    // Closest point on the box
    qPrime.x = box.extents.x;
    qPrime.x = line.origin.y;
    qPrime.x = line.origin.z;
    //
    // Compute distance squared and Y and Z components
    // of box's closest point
    //
    if (line.origin.y < - box.extents.y) {
        delta = line.origin.y + box.extents.y;
        distanceSquared += delta * delta;
        qPrime.y = -box.extents.y;
    } else if (line.origin.y > box.extents.y) {
        delta = line.origin.y - box.extents.y;
        distanceSquared += delta * delta;
        qPrime.y = box.extents.y;
    }

    if (line.origin.z < - box.extents.z) {
        delta = line.origin.z + box.extents.z;
        distanceSquared += delta * delta;
        qPrime.z = -box.extents.z;
    } else if (line.origin.z > box.extents.z) {
        delta = line.origin.z - box.extents.z;
        distanceSquared += delta * delta;
        qPrime.z = box.extents.z;
    }

    return distanceSquared;
}
```

3. 一个零分量

在这种情形中，变换后的直线垂直于一个基底向量。图 10.49 显示了垂直于有向有界箱的 z 轴的一条直线，它可能在有向有界箱的上面、里面或下面。注意，在这种情形中，直线与有向有界箱不相交，最接近的点总是位于一个角上。

可以很好地优化这种情形的实现代码。这里假设（变换后的）直线的方向向量 $\vec{d}'$ 仅有正分量。在这种情形中，如果直线与有向有界箱不相交，那么最接近的点将位于有向有界箱的左上角或右下角，如图 10.49 所示。通过观察从右上角到直线原点的向量 $\vec{e}$ 与直线的方向向量 $\vec{d}'$ 之间的夹角，我们可以确定可能是哪种情形。7.1 节讨论的 Kross 函数可以应用于这两个向量，以确定它们之间的夹角是正的还是负的。如果 $\text{Kross}(\vec{e},\vec{d}')>0$，那么直线将与“$x$ 轴”相交；否则，它将与“y 轴”相交。在前一种情形中，有向有界箱上最接近的点（如果不相交）将位于右下角，而在后一种情形中，最接近的点将位于左上角。图 10.50 示意了这一点。确定这一点后，我们还需要确定直线与有向有界箱是否相交。考虑右下角可能是最接近点的情形：如果直线与从最接近的角到直线原点的向量之间的夹角是正的，那么直线与有向有界箱相交；如果夹角为负，那么直线与有向有界箱不相交。由于余弦函数是关于 0 对称的，因此这两个向量之间的点积并不能说明问题。然而，我们可以对直线

与角到直线原点的向量的“垂直运算”应用 Kross 函数，如图 10.51 所示。如果直线与有向有界箱相交，那么距离为零；否则，距离平方就是Q'与变换后的直线之间的距离平方。

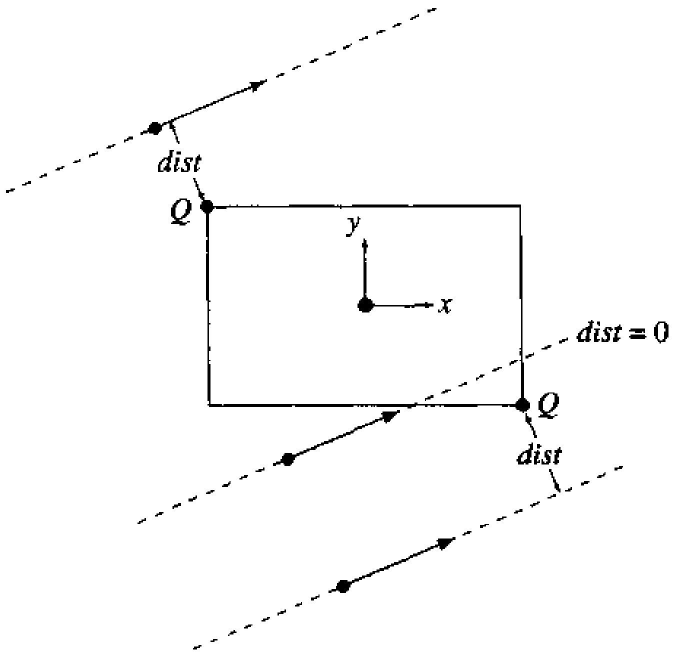


图 10.49　一个零分量的情形

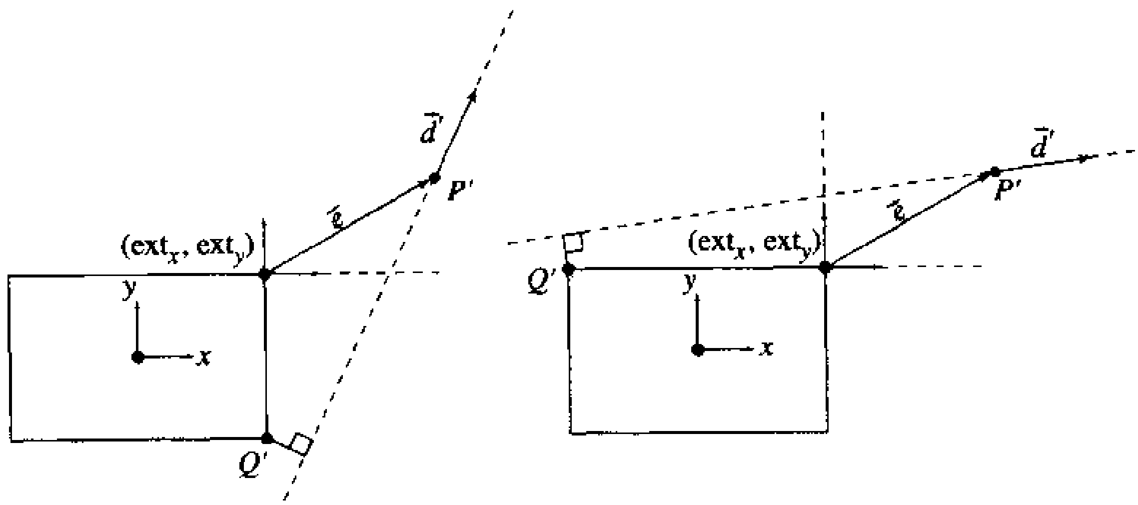


图 10.50　确定在何处寻找箱上的最近点

伪码为

```
real CaseOneZeroComponent_Z(Line line, OBB box, real t0, Point qPrime)
{
    Vector3D ptMinusExtents = line.origin - box.extents;
    real prod0 = line.direction.y * ptMinusExtents.x;
    real prod1 = line.direction.x * ptMinusExtents.y;
    real distanceSquared = 0;
    qPrime.z = line.origin.z;
```

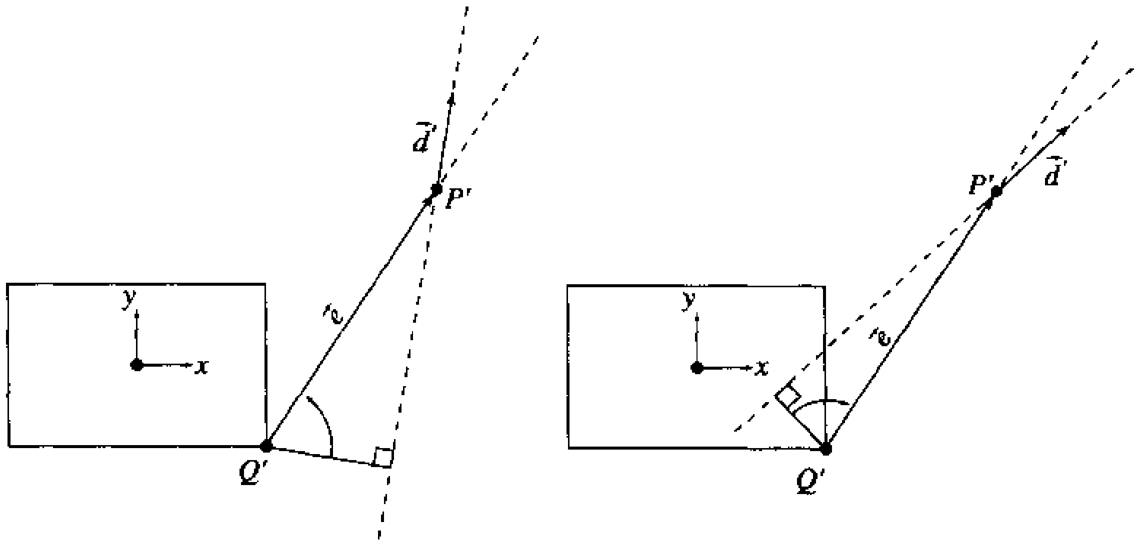


图 10.51　确定直线与箱是否相交

```
    if (prod0 >= prod1) {
        //
        // line intersects ''x-axis'' of OBB
        // Closest point is along bottom edge of right face of OBB
        //
        qPrime.x = box.extent.x;
        real tmp = line.origin.y + box.extents.y;
        delta = prod0 - line.direction.x * tmp;
        if (delta >= 0) {
            // There is no intersection, so compute distance
        invLSquared = 1 / (line.direction.x * line.direction.x +
                              line.direction.y + line.direction.y);
        distanceSquared += (delta * delta) * invLSquared;

        // If desired, compute the parameter of the line's closest
        // point, and set the closest point on the box to be along
        // the lower-right edge.
        qPrime.y = -box.extents.y;
        t0 = -(line.direction.x * ptMinusExtents.x * line.direction.y + tmp)
                * invLSquared;
    } else {
        // Line intersects box. Distance is zero.
        inv = 1 / line.direction.x;
        qPrime.y = line.origin - (prod0 * inv);
        t0 = -ptMinusExtents.x * inv;
    }
} else {
    //
    // line intersects the ''y-axis'' of OBB
    // Closest point is along top edge of left face of OBB
    // (or, equivalently, left edge of top face)
```

```
        // Code exactly parallels that above...

    }

        // Now, consider the z-direction
        if (line.origin.z < -box.extents.z) {
            delta = line.origin.z + box.extents.z;
            distanceSquared += delta * delta;
            qPrime.z = -box.extents.z;
        } else if (line.origin.z > box.extents.z) {
            delta = line.origin.z - box.extents.z;
            distanceSquared += delta * delta;
            qPrime.z = box.extents.z;
        }

        return distanceSquared;
    }
```

4. 没有零分量

在这种情形中，变换后的直线与任何的基底向量不平行。然而，我们可以像在“一个零分量”的情形中那样使用 Kross 运算，并假设变换后的直线的所有分量都是正的。这样，我们就能确定直线将首先与哪个平面相交，并在确定直线与有向有界箱的哪个面相交或者直线与有向有界箱的某个面的距离时利用这一信息。

伪码为

```
real CaseNoZeroComponents(Line line, OBB box, real t, Point qPrime)
{
    Vector3D ptMinusExtents = line.origin - box.extents;
    real dyEx = line.direction.y * ptMinusExtents.x;
    real dxEy = line.direction.x * ptMinusExtents.y;

    if (dyEx >= dxEy) {
        real dzEx = line.direction.z * ptMinusExtents.x;
        real dxEz = line.direction.x * ptMinusExtents.z;

        if (dzEx >= dxEz) {
            // line intersects x = box.extent.x plane
            distanceSquared = FaceX(line, box, ptMinusExtents, t, qPrime);
        } else {
            // line intersects z = box.extent.z plane
            distanceSquared = FaceZ(line, box, ptMinusExtents, t, qPrime);
        }
    } else {
        real dzEy = line.direction.z * ptMinusExtents.y;
        real dyEz = line.direction.y * ptMinusExtents.z;

        if (dzEy >= dyEz) {
            // line intersects y = box.extent.y plane
            distanceSquared = FaceY(line, box, ptMinusExtents, t, qPrime);
        } else {
```

```
            // line intersects z = box.extent.z plane
            distanceSquared = FaceZ(line, box, ptMinusExtents, t, qPrime);
        }
    }

    return distanceSquared;
}
```

图 10.52 显示了每一个"正"面，我们将测试直线与它们的相交和距离。直线将与该面所在的平面相交（它可能最先与面相交也可能不会）。由于直线方向向量的所有分量都为正，因此直线不可能与这个面的四条边中的两条最接近（在图中，可能是最接近的边显示为较粗的线）。因此，我们仅仅对该面的边所定义的 9 个区域中的 6 个感兴趣。

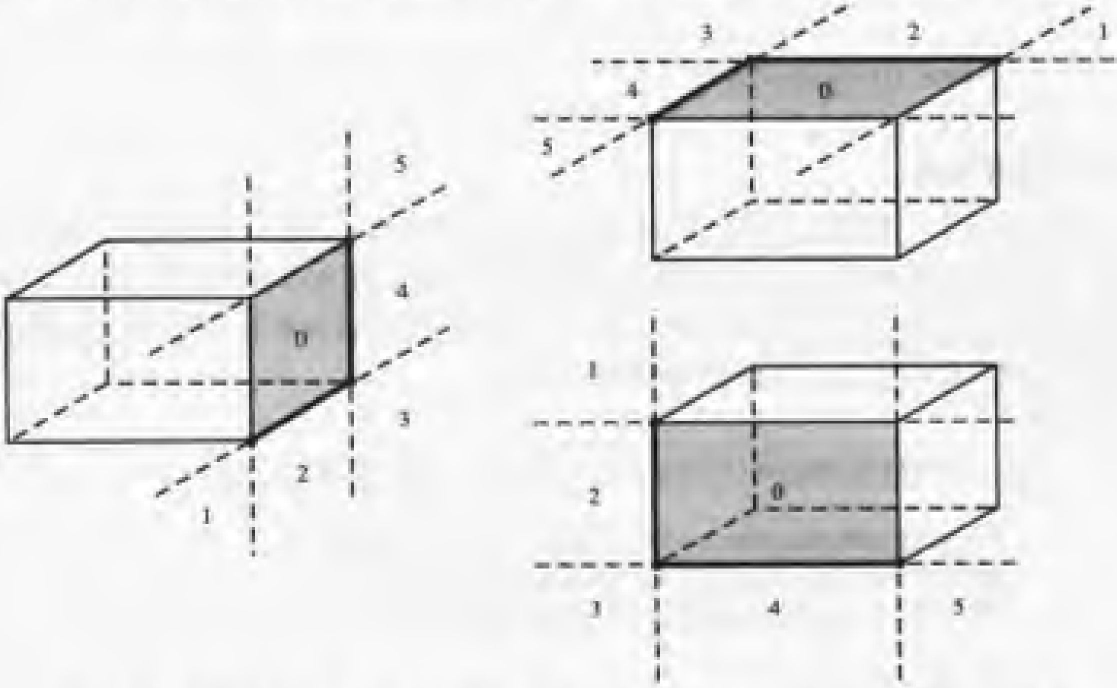


图 10.52　具有两条边和三个顶点的 OBB 的每个"正"面都可能最接近于指定的直线

对"正"的 X 面进行面测试的过程的伪码如下：

```
real Face_X(Line line, OBB box, Vector3D ptMinusExtents, real t, Point3D qPrime)
{
    Point3D qPrime = line.origin;
    real    distanceSquared = 0;

    Vector3D ptPlusExtents = line.origin + box.extents;
    if (line.direction.x * ptPlusExtents.y >= line.direction.y * ptMinusExtents.x) {
        //
        // region 0, 5, or 4
        //
        if (line.direction.x * pPe.z >= line.direction.z * ptMinusExtents.x) {
            //
            // region 0 - line intersects face
            //
            qPrime.x = box.extents.x;
```

```
        inverse = 1.0 / line.direction.x;
        qPrime.y -= line.direction.y * ptMinusExtents.x * inverse;
        qPrime.z -= line.direction.z * ptMinusExtents.x * inverse;
        t = -ptMinusExtents * inverse;
    } else {
        //
        //  region 4 or 5
        //
        lSqr = line.direction.x * line.direction.x
               + line.direction.z * line.direction.z;
        tmp = lSqr * ptPlusExtents.y - line.direction.y
              * (line.direction.x * ptMinusExtents.x + line.direction.z
                                   * ptPlusExtents.z);
        if (tmp <= 2 * lSqr * box.extents.y) {
            //
            //  region 4
            //
            tmp = ptPlusExtents.y - (tmp / lSqr);
            lSqr += line.direction.y * line.direction.y;
            delta = line.direction.x * ptMinusExtents.x + line.direction.y * tmp
                    + line.direction.z * ptPlusExtents.z;

            t = -delta / lSqr;
            distanceSquared += ptMinusExtents.x * ptMinusExtents.x
                               + tmp * tmp
                               + ptPlusExtents.z * ptPlusExtents.z
                               + delta * t;

            qPrime.x = box.extents.x;
            qPrime.y = t - box.extents.y;
            qPrime.z = -box.extents.z;
        } else {
            //
            //  region 5
            //
            lSqr += line.direction.y * lineDirection.y;
            delta = line.direction.x * ptMinusExtents.x
                    + line.direction.y * ptMinusExtents.y
                    + line.direction.z * ptPlusExtents.z;
            t = -delta / lSqr;
            distanceSquared += ptMinusExtents.x * ptMinusExtents.x
                               + ptMinusExtents.y * ptMinusExtents.y
                               + ptPlusExtents.z * ptPlusExtents.z
                               + delta * t;

            qPrime.x =  box.extents.x;
            qPrime.y =  box.extents.y;
            qPrime.z = -box.extents.z;
        }
    }
} else {
```

```
if (line.direction.x * ptPlusExtents.z >=  line.direction.z * ptMinusExtents.x)
    //
    //  region 1 or 2
    //
    lSqr = line.direction.x * line.direction.x + line.direction.y
          * line.direction.y;
    tmp = lSqr * ptPlusExtents.z - line.direction.z * (line.direction.x
         * ptMinusExtents.x + line.direction.i1 * ptPlusExtents.y);
    if (tmp <= 2 * lSqr * box.extents.z) {
        //
        //  region 2
        //
        tmp = ptPlusExtents.z - (tmp / lSqr);
        lSqr += line.direction.z * line.direction.z;
        delta = line.direction.x * ptMinusExtents.x + line.direction.y
               * ptPlusExtents.y + line.direction.z * tmp;
        t = -delta / lSqr;
        distanceSquared += ptMinusExtents.x * ptMinusExtents.x
                         + ptMinuxExtents.y * ptMinusExtents.y
                         + tmp * tmp
                         + delta * t;

        qPrime.x = box.extents.x;
        qPrime.y = -box.extents.y;
        qPrime.z = t - box.extents.z;
    } else {
        //
        //  region 1
        //
        lSqr += line.direction.z * lineDirection.z;
        delta = line.direction.x * ptMinusExtents.x
               + line.direction.y * ptMinusExtents.y
               + line.direction.z * ptPlusExtents.z;
        t = -delta / lSqr;
        distanceSquared += ptMinusExtents.x * ptMinusExtents.x
                         + ptPlusExtents.y * ptPlusExtents.y
                         + ptMinusExtents.z * ptMinusExtents.z
                         + delta * t;

        qPrime.x =  box.extents.x;
        qPrime.y = -box.extents.y;
        qPrime.z =  box.extents.z;
    }
} else {
    lSqr = line.direction.x * line.direction.x +
           line.direction.z * line.direction.z;
    tmp = lSqr * ptPlusExtents.y - line.direction.y * (line.direction.x
```

```
         * ptMinusExtents.x + line.direction.z * ptPlusExtents.z);
    if (tmp >= 0) {
        //
        //  region 4 or 5
        //
        if (tmp <= 2 * lSqr * box.extents.y) {
            //
            //  region 4. Code block same as previous region 4 code.
            //
        } else {
            //
            //  region 5. Code block same as previous region 5 code.
            //
        }
    }

    lSqr = line.direction.x * line.direction.x + line.direction.y
           * line.direction.y;
    tmp = lSqr * ptPlusExtents.z
          - line.direction.z * (line.direction.x * ptMinusExtents.x
                                + line.direction.y * ptPlusExtents.y);
    if (tmp >= 0) {
        //
        //  region 1 or 2
        //
        if (tmp <= 2 * lSqr * box.extents.z) {
            //
            //  Region 2. Code block same as previous region 2 code.
            //
        } else {
            //
            //  Region 1. Code block same as previous region 1 code.
            //
        }
        return distanceSquared;
    }
        //
        //  region 3
        //
        lSqr += line.direction.y * line.direction.y;
        delta = line.direction.x * ptMinusExtents.x
                + line.direction.y * ptPlusExtents.y
                + line.direction.z * ptPlusExtents.z;
        t = -delta / lSqr;
        distanceSquared +=
            ptMinusExtents.x * ptMinusExtents.x
            + ptPlusExtents.y * ptPlusExtents.y
            + ptPlusExtents.z * ptPlusExtents.z
            + delta * t;
        qPrime.x =  box.extents.x;
        qPrime.y = -box.extents.y;
```

```
                qPrime.z = -box.extents.z;
            }
        }
    }
```

5. 射线到有向有界箱的距离

在计算射线到有向有界箱的距离时，我们首先计算射线所在的无限直线与有向有界箱的距离。如果射线上最接近的点的参数t小于零，那么，我们就必须计算射线的端点到有向有界箱的距离。

伪码为

```
real RayOBBDistanceSquared(Ray ray, OBB box)
{
    Line line;
    line.origin = ray.origin;
    line.direction = ray.direction;

    distanceSquared = LineOBBDistanceSquared(line, box, t);
    if (t < 0) {
        distanceSquared = PointOBBDistanceSquared(ray.origin, box);
    }

    return distanceSquared;
}
```

6. 线段到有向有界箱的距离

在计算线段到有向有界箱的距离时，我们首先计算线段所在的无限直线与有向有界箱的距离。如果线段上最接近的点的参数t不位于[0, 1]内，那么，如果$t < 0$，我们就必须计算线段的起点到有向有界箱的距离；如果$t > 1$，我们就必须计算线段的终点到有向有界箱的距离。

伪码为

```
real LineSegmentOBBDistanceSquared(Segment seg, OBB box)
{
    Line line;
    line.origin = seg.start
    line.direction = seg.end - seg.start;

    distanceSquared = LineOBBDistanceSquared(line, box, t);
    if (t < 0) {
        distanceSquared = PointOBBDistanceSquared(seg.start, box);
    } else if (t > 1) {
        distanceSquared = PointOBBDistanceSquared(seg.end, box);
    }

    return distanceSquared;
}
```

10.10 直线到二次曲面的距离

如果直线与二次曲面相交，那么它们之间的距离为零。可以使用直线的参数形式，即 $X(t)=P+t\vec{d}$，来进行相交测试。二次曲面用 $Q(X)=X^{\mathrm{T}}\mathrm{A}X+B^{\mathrm{T}}X+c=0$ 来隐式表示。将直线方程代入二次曲面方程，可以得到如下的多项式方程

$$(\vec{d}^{\mathrm{T}}\mathrm{A}\vec{d})t^2+\vec{d}^{\mathrm{T}}(2\mathrm{A}P+B)t+(P^{\mathrm{T}}\mathrm{A}P+B^{\mathrm{T}}P+c)=e_2t^2+e_1t+e_0=0$$

当 $e_1^2-4e_0e_2\geqslant 0$ 时，该方程有实数解，此时直线与二次曲面之间的距离为零。

如果方程仅有复数解，那么直线与二次曲面不相交且它们之间的距离为正。首先考虑一个圆柱体，它的轴就是上述的直线，它的半径足够小使得它与曲面并不相交。随着半径的增加，柱体最后将刚好与曲面接触，一般接触于一个点，但也可能沿着整条的直线。直线与曲面之间的距离就是该柱体的半径，并对应于位于柱体上的曲面点与柱体的轴之间的距离。如果直线表示为两个平面之间的交线，两个平面为 $\hat{n}_i\cdot X=c_i=\hat{n}_i\cdot P$，其中 $i=0,1$，$\{\vec{d},\hat{n}_0,\hat{n}_1\}$ 是一组互相正交的向量，并且 $\|\hat{n}_i\|=1$，那么从 X 到柱体轴之间的距离平方为 $F(X)=(\hat{n}_0\cdot X-c_0)^2+(\hat{n}_1\cdot X-c_1)^2$。问题转化为在曲面上找到一个点 X 使得 $F(X)$ 取得最小值。这是一个有约束条件的最小值问题，可以使用拉格朗日乘子来求解，求解方法如下。

定义

$$G(X,s)=(\hat{n}_0\cdot X-c_0)^2+(\hat{n}_1\cdot X-c_1)^2+sQ(X)$$

其中 s 是作为乘子而引入的参数。当 $\nabla Q=\vec{0}$ 且 $\partial G/\partial s=0$ 时，G 取得最小值。第一个方程为 $2(\hat{n}_0\cdot X-c_0)\hat{n}_0+2(\hat{n}_1\cdot X-c_1)\hat{n}_1+s\nabla Q=\vec{0}$，应用约束条件 $Q=0$ 就可得到第二个方程。将第一个方程与 $\hat{d}$ 点乘，可得如下的条件

$$L(X):=\vec{d}\cdot\nabla Q(X)=0$$

这是一个关于 X 的线性方程。在几何意义上，$\vec{d}\cdot\nabla Q=0$ 意味着当最小距离为正时，连接两个最接近点的线段必定同时垂直于直线和二次曲线。参见说明二维设置的图 6.23。现在将第一个方程与 $\hat{n}_0$ 和 $\hat{n}_1$ 点乘，得到两个关于 s 的方程，即 $2(\hat{n}_1\cdot X-c_1)+s\hat{n}_0\cdot\nabla Q=0$ 和 $2(\hat{n}_0\cdot X-c_0)+s\hat{n}_1\cdot\nabla Q=0$。为了使两个方程都具有正常解，必须有

$$M(X):=(\hat{n}_0\cdot X-c_0)\hat{n}_1\cdot\nabla Q-(\hat{n}_1\cdot X-c_1)\hat{n}_0\cdot\nabla Q=0$$

这是一个关于 X 的二次方程。

我们得到了一个系统，它包含三个关于 X 的三个位置分量的多项式方程。其中两个方程 $Q=0$ 和 $M=0$ 是二次方程，而第三个方程 $L=0$ 是线性方程。A.2 节说明了如何求解这样的系统。与计算直线与椭圆之间距离的二维问题一样，必须小心地求解系统。当 $\mathrm{A}\vec{d}=\vec{0}$ 时，$L=0$ 退化，此时方程成为 $\vec{d}\cdot B=0$。同时注意，方程 $M=0$ 可改写为 $\hat{n}\cdot(2\mathrm{A}X+B)=0$，其中 $\hat{n}=(\hat{n}_0\cdot X-c_0)\hat{n}_1-(\hat{n}_1\cdot X-c_1)\hat{n}_0$ 是 $\vec{d}$ 的正交向量。如果 $\mathrm{A}\hat{n}=\vec{0}$，那么方程 $M=0$ 退化为 $\hat{n}\cdot B=0$。当二次曲面包含直线时，例如，一个抛物柱面、双曲柱面或双曲面，可能出现上述的一种退化情形。实现代码必须能处理这种退化情形，并调用合适的代码段来计算直线之间的距离。

计算直线与二次曲面之间距离的另一种方法是利用求解最小值的算法。如果直线为

$X(t)=P+t\vec{d}$（$t\in\mathbb{R}$），而且点X与二次曲面之间的距离为$F(X)$，那么直线上的点$X(t)$与二次曲面之间的距离为$G(t)=F(P+t\vec{d})$。通过搜索t的定义域$\mathbb{R}$以确定可得到$G(t)$的最小值的t，就能实现求解最小值的算法。需要考虑两方面的折中。如果进行消元以得到一个高次数的多项式方程，这种建立多项式方程的方法可能存在数值问题。由于存在接近于零的系数，消元过程和求根过程很可能产生数值误差。建立函数并求最小值的方法在数值上可能更稳定。然而，如果初始估值不接近于最小值点或者迭代陷于局部最小值而不是全局最小值，收敛于最小值的方法将是一种很慢的方法。

10.11 直线到多项式曲面的距离

设直线可表示为$\mathcal{L}(t)=P+t\vec{d}$（$t\in\mathbb{R}$且$\|\vec{d}\|=1$）。设曲面用参数形式表示为

$$X(r,s)=\sum_{i_0=0}^{n_0}\sum_{i_1=0}^{n_1}A_{i_0,i_1}r^{i_0}s^{i_1}$$

其中$(r,s)\in[r_{\min},r_{\max}]\times[s_{\min},s_{\max}]$。一对点（一个在曲面上，另一个在直线上）之间的距离平方为

$$F(r,s,t)=\|X(r,s)-\mathcal{L}(t)\|^2$$

曲面和直线之间的距离平方就是$\min\{F(r,s,t):(r,s,t)\in[r_{\min},r_{\max}]\times[s_{\min},s_{\max}]\times\mathbb{R}\}$。可对$F$直接应用最小值求解方法，但一般要注意如下的问题：小心收敛于一个局部最小值而不是全局最小值。计算最小值的标准微积分方法是建立一个临界点集，对该集内的每一点计算F的值，并选择这些F值中的最小值。F的一阶导数为$F_r=2(X-\mathcal{L})\cdot X_r$，$F_s=2(X-\mathcal{L})\cdot X_s$和$F_t=-(X-\mathcal{L})\cdot\vec{d}$，其中$X_r$和$X_s$为$X$的一阶导数。注意$X$的次数为$n_0+n_1$，$X_r$和$X_s$的次数为$n_0+n_1-1$。临界点可定义如下：

（1）$(F_r,F_s,F_t)(r,s,t)=(0,0,0)$，其中，$(r,s,t)\in(r_{\min},r_{\max})\times(s_{\min},s_{\max})\times\mathbb{R}$

（2）$(F_r,F_t)(r,s_{\min},t)=(0,0)$，其中，$(r,t)\in(r_{\min},r_{\max})\times\mathbb{R}$

（3）$(F_r,F_t)(r,s_{\max},t)=(0,0)$，其中，$(r,t)\in(r_{\min},r_{\max})\times\mathbb{R}$

（4）$(F_s,F_t)(r_{\min},s,t)=(0,0)$，其中，$(s,t)\in(s_{\min},s_{\max})\times\mathbb{R}$

（5）$(F_s,F_t)(r_{\max},s,t)=(0,0)$，其中，$(s,t)\in(s_{\min},s_{\max})\times\mathbb{R}$

（6）$F_t(r_{\min},s_{\min},t)=0$，其中，$t\in\mathbb{R}$

（7）$F_t(r_{\min},s_{\max},t)=0$，其中，$t\in\mathbb{R}$

（8）$F_t(r_{\min},s_{\min},t)=0$，其中，$t\in\mathbb{R}$

（9）$F_t(r_{\max},s_{\max},t)=0$，其中，$t\in\mathbb{R}$

上表中第一项要求求解三个具有三个未知数r，s和t的方程。定义$\vec{\Delta}(r,s)=X(r,s)-P$。可针对$t=\vec{d}\cdot\vec{\Delta}$求解方程$F_t=0$。将其代入方程$F_r=0$和$F_s=0$，得到多项式方程$(\vec{\Delta}-(\vec{d}\cdot\vec{\Delta})\vec{d})\cdot X_r=0$和$(\vec{\Delta}-(\vec{d}\cdot\vec{\Delta})\vec{d})\cdot X_s=0$。第一个方程中$r$的次数为$2n_0-1$，$s$的次数为$2n_1$。第二个方程中$r$的次数为$2n_0$，$s$的次数为$2n_1-1$。消元理论（Wee 和 Goldman，1995a，1995b）可以用来将这一过程简化为一个关于r的高次数多项式方程$G(r)=0$。对方程的每一个根$\bar{r}$，对应的值$\bar{s}$为$(\vec{\Delta}-(\vec{d}\cdot\vec{\Delta})\vec{d})\cdot X_r=0$的根。对每一个这样的$(\bar{r},\bar{s})$对，

计算对应的 $\bar{t}$ 值，并计算 $F(\bar{r},\bar{s},\bar{t})$ 的值。在整个过程中都记录最小的 F 值。

上表中第二项要求求解两个具有两个未知数 r 和 t 的方程。也针对 $t=\vec{d}\cdot\vec{\Delta}$ 求解方程 $F_t=0$，只是现在 $\vec{\Delta}$ 只随 r 变化，因为 $s=s_{\min}$。将其代入方程 $F_r=0$，得到一个关于 r 的次数为 $2n_0-1$ 的多项式方程。对方程的每一个根 $\bar{r}$，计算对应的 $\bar{t}$ 值，并计算 $F(\bar{r},s_{\min},\bar{t})$ 的值。在整个过程中，如果计算得到的新值小于原来原来记录的最小的 F 值，那么更新最小的 F 值记录。项目 3，项目 4 和项目 5 的处理与此类似。

项目 6 的处理很简单。对 $\bar{t}=\vec{d}\cdot\vec{\Delta}$ 求解方程，计算 $F(r_{\min},s_{\min},\bar{t})$，并用该值来做必要的对当前存储的 F 的最小值的更新。项目 7，项目 8 和项目 9 的处理与此类似。

10.12 GJK 算法

计算两个凸多面体之间的距离的GJK 算法与其二维空间中的对应问题（即计算两个凸多边形之间的距离）具有相同的理论基础。6.10 节中的材料用包含于凸对象内的单形来描述 n 维问题，并取 $n=3$。自然，在该维中，单形就是四面体。一个已实现的叫做 SOLID（Software Library for Interference Detection）的三维碰撞检测系统是基于三维 GJK 算法而建立的，Van den Bergen（1997）描述了该系统。Van den Bergen（1997，1999，2001a）提供了这一实现所采用的不同算法的描述。这个实现最引人瞩目的地方是它特别注意了数值问题，而数值问题正是其他GJK实现的灾难。

10.13 杂项

本节包含了处理一些特殊类型的距离问题的算法。特别介绍的问题包括：（1）一条直线与一条平面曲线之间的距离；（2）两条不位于同一平面上的平面曲线之间的距离；（3）运动物体之间的距离；（4）一个曲面上的两个点之间的距离。最后的一个问题要求计算两个点之间的最短路径，而且路径本身也必须位于曲面上。这种路径叫做测地路径（geodesic path），其长度叫做测地距离（geodesic distance）。

10.13.1 直线与平面曲线之间的距离

本节介绍了计算直线与平面曲线之间的距离的两种类型的方法。第一种方法基于曲线的参数表示法。第二种方法基于将曲线表示为一些代数方程所表示的系统的解集。在两种情形中，直线都用参数形式表示为 $X(t)=P+t\hat{d}$，其中 $\|\hat{d}\|=1$，并且曲线所在的平面为 $\hat{n}\cdot X=c$，其中 $\|\hat{n}\|=1$。

1. 参数表示法

设对于某些定义域 $[s_{\min},s_{\max}]$，曲线可以用参数形式表示为 $X(s)$。一对点（一个位于曲线上，另一个位于直线上）之间的距离平方为 $F(s,t)=\|X(s)-(P+t\hat{d})\|^2$。要计算直线与曲线之间的最小距离，就要计算出使 F 取最小值的一个参数对 $(s,t)\in[s_{\min},s_{\max}]\times\mathbb{R}$。可以直接对 F 应用求最小值的算法，特别是当 X 关于 s 不是连续可微时。如果导数存在且连续，那么可以使用一种微积分方法。特别地，当满足下列条件之一时，将出现最小值：

i. $\nabla F(s,t)=\vec{0}$ 对于一个 $(s,t)\in(s_{\min},s_{\max})\times\mathbb{R}$ 成立

ii. $\partial F(s_{\min},t)/\partial t=0$ 对于某些 $t\in\mathbb{R}$ 成立

iii. $\partial F(s_{\max},t)/\partial t=0$ 对于某些 $t\in\mathbb{R}$ 成立

要求解的最简单的偏导数方程是 $\partial F/\partial t=0$。其解为 $t=\hat{d}\cdot(X(s)-P)$。如果使用完整的梯度，那么可以替换另一个导数方程中的 t 值

$$0=\frac{\partial F}{\partial s}=((X(s)-P)-(\hat{d}\cdot(X(s)-P))\hat{d})\cdot X'(s)$$

$$=X'(s)^{\mathrm{T}}(I-\hat{d}\hat{d}^{\mathrm{T}})(X-P) \tag{10.16}$$

求解该方程的复杂程度与曲线本身的复杂程度直接相关。

【实例】 直线与圆之间的距离。一个三维空间上的圆可以用圆心 C，半径 r，以及包含该圆的平面 $\hat{n}\cdot(X-C)=0$ 来定义，其中 $\hat{n}$ 是平面的单位长度法线。如果 $\hat{u}$ 和 $\hat{v}$ 也是单位长度向量，且 $\hat{u}$，$\hat{v}$ 和 $\hat{n}$ 构成一个正交集，那么圆可以用参数方式表示为

$$X(s)=C+r(\cos(s)\hat{u}+\sin(s)\hat{v})=:C+r\hat{w}(s)$$

其中 $s\in[0,2\pi)$。所有 X 的值都与 C 等距，因为 $\|X-C\|=r$，且 $\|\hat{w}\|=1$。因为 $\hat{u}$ 和 $\hat{v}$ 都垂直于 $\hat{n}$，因此 $\hat{n}\cdot(X-C)=0$，因而它们都位于同一平面内。

设 $\vec{\Delta}=C-P$，经过一些代数运算，针对该例的方程（10.16）可变化为

$$a_{10}\cos s+a_{01}\sin s+a_{20}\cos^2 s+a_{11}\cos s\sin s+a_{02}\sin^2 s=0$$

其中 $a_{10}=\vec{\Delta}\cdot\hat{v}-(\hat{d}\cdot\hat{v})(\hat{d}\cdot\vec{\Delta})$，$a_{01}=-\vec{\Delta}\cdot\hat{u}+(\hat{d}\cdot\hat{u})(\hat{d}\cdot\vec{\Delta})$，$a_{20}=-r(\hat{d}\cdot\hat{u})(\hat{d}\cdot\hat{v})$，$a_{11}=r[(\hat{d}\cdot\hat{u})^2-(\hat{d}\cdot\hat{v})^2]$，且 $a_{02}=r(\hat{d}\cdot\hat{u})(\hat{d}\cdot\hat{v})$。可以对该方程应用一种数值根的求解方法。每一个 $\bar{s}$ 都用于计算 $\bar{t}=\hat{d}\cdot(X(\bar{s})-P)$，并计算一个对应的距离平方 $F(\bar{s},\bar{t})$。直线与圆之间的距离平方就是用所有根计算得到的 F 值的最小值。用这种方法计算距离所需的 CPU 时间可能并不比直接使用数值方法来求 F 的最小值所需的时间少。在两种方法中，计算所花费时间的大部分都用于计算三角函数值。■

2. 代数表示法

设曲线定义为包含它的平面和一个用多项式方程 $F(X)=0$ 所隐式定义的曲面之间的交线。假定 ∇F 与平面的法线 $\hat{n}$ 不平行，这是一种合理的假设，因为只有曲面横切平面才能形成曲线。点 X 到直线的垂直距离线段 $S(X)=(X-P)-(\hat{d}\cdot(X-P))\hat{d}$ 的长度。该线段是 $X-P$ 在正交于 $\hat{d}$ 的平面上的投影。现在的问题是计算点 X，使它在具有约束条件 $\hat{n}\cdot X=c$ 并且当 $F(X)=0$ 时满足距离平方 $\|S(X)\|^2$ 取最小值。A.9.3 节讨论的拉格朗日乘子法可用来求解这类问题。我们已知

$$\|S\|^2=(X-P)^{\mathrm{T}}(I-\hat{d}\hat{d}^{\mathrm{T}})(X-P)=(X-P)^{\mathrm{T}}\mathrm{A}(X-P)$$

其中 $\mathrm{A}=\mathrm{I}-\hat{d}\hat{d}^{\mathrm{T}}$。定义

$$G(X,u,v)=(X-P)^{\mathrm{T}}\mathrm{A}(X-P)+u(\hat{n}\cdot X-c)+vF(X)$$

参数 u 和 v 都是拉格朗日乘子。当 $\partial G/\partial X=\vec{0}$，$\partial G/\partial u=0$ 并且 $\partial G/\partial v=0$ 时，函数 G 具有最小值。表达式 $\partial G/\partial X$ 表示 G 对于 X 的三个分量的第一阶偏导数所构成的三元组。最后

的两个方程只是重述了定义该对象的两个约束条件。根据假设，$\hat{n}\cdot\nabla F\neq\vec{0}$，因此，用该向量点乘$X$的导数方程，再除以 2 将产生如下的数量方程

$$H(X):=\hat{n}\times\nabla F\cdot \mathrm{A}(X-P)=0$$

使距离取最小值的可能的点X就是如下多项式方程定义的系统的解：$\hat{n}\cdot X=c$，$F(X)=0$，以及$H(X)=0$。

【实例】直线与圆之间的距离。考虑前面讨论过的例子。F具有多种选择。例如，F可以定义一个半径为r，球心为C的球面。或者F可以定义一个圆心为C，具有方向$\hat{n}$，半径为r的柱面。球面的方程较简单，因此我们采用球面，即$F(X)=\|X-C\|^2-r^2$和$\nabla F=2(X-C)$。函数$H(X)$是二次多项式。将消元理论应用于一个线性方程和两个二次方程将得到一个高次数的多项式。我们使用平面坐标而不是直接求解这三个方程。设$X=C+u\hat{u}+v\hat{v}$，其中$\{\hat{u},\hat{v},\hat{n}\}$是一个正交集。$X$的这种表示法自动满足平面方程$\hat{n}\cdot X=c$，其中$c=\hat{n}\cdot C$。将$X$代入$F=0$和$H=0$，将得到两个具有两个未知数$u$和$v$的二次方程$f(u,v)=0$和$h(u,v)=0$。对$v$进行消元，得到一个二次方程$g(u)=0$。对每一个根$\bar{u}$，通过求解二次方程$f(\bar{u},v)=0$，可计算出两个$\bar{v}$值。计算每一对（$\bar{u}$，$\bar{v}$）的距离平方，从这些距离平方值中选取最小的距离平方值。■

10.13.2　直线与平面实心物体之间的距离

上述的方法用于计算直线与曲线之间的距离。如果要求直线与一个边界为指定曲线的平面实心对象之间的距离，那么上述算法必须做一点修改。如果直线与平面相交于实心对象内的一点，那么距离为零。如果直线与平面相交，但与实心对象不相交，那么实心对象上最接近于直线的点必定是对象边界上的点。证明很简单。如果最接近的点是一个内点，那么它必定是通过直线上最接近的点投影到对象所在的平面上所得到的点。起始于直线上最接近的点，朝着平面沿着直线走足够的一小步，得到一个新的点，其投影成为对象的另一个内点，并且新的点对之间的距离比原来的点对之间的距离更小，而这与原来的点对之间的距离是最小距离相矛盾。最后，如果直线与平面平行，那么对象上最接近于直线的点可从物体的边界上得到。因此，算法取决于直线与对象是否相交。如果相交，距离为零；否则距离为直线与对象边界之间的距离。

10.13.3　平面曲线之间的距离

如果两条平面曲线都位于同一三维平面$\hat{n}\cdot(X-C)=0$上，那么它们之间的距离可用平面坐标来计算，这可以有效地将问题简化为一个二维问题。也就是说，如果$\hat{n}$是单位长度向量，$\hat{u}$和$\hat{v}$也是单位长度向量，并满足$\{\hat{u},\hat{v},\hat{n}\}$是一个正交集，那么平面上的任何点都可表示为$X=C+r\hat{u}+s\hat{v}+t\hat{n}$，其中$r$，$s$，$t$为数量。通过投影法线分量来计算平面内的距离。是平面的单位长度法线。投影点为$C+r\hat{u}+s\hat{v}$。可以利用二维距离算法来计算它与点(r,s)之间的距离。在三维空间中，点C对应于二维空间中的原点（0，0）。

应用程序可能要求计算不在同一平面上的两条平面曲线之间的距离。在这种情形中，问题需要在完整的三维空间中进行计算。我们提供了两种类型的方法来处理这样的曲线。

如果平面曲线用参数形式来表示，那么可以应用其中的一种方法。如果平面曲线的点集表示为代数方程系统的解集，那么可以应用另一种方法。

1. 参数表示法

设对于某些定义域$[s_{\min}, s_{\max}]$，曲线可以用参数形式表示为$X(s)$，并且对于某些定义域$[t_{\min}, t_{\max}]$，曲线可以用参数形式表示为$Y(t)$。一对点（各位于一条曲线上）之间的距离平方为$F(s,t)=\|X(s)-Y(t)\|^2$。要计算两条曲线之间的最小距离，就要计算出使F取最小值的一个参数对$(s,t)\in[s_{\min}, s_{\max}]\times[t_{\min}, t_{\max}]$。可以直接对$F$应用求最小值的算法，特别是当$X$或$Y$不是连续可微时。如果两条曲线都是连续可微的，那么可以使用微积分方法。最小值确定地出现在$\nabla F=\vec{0}$的一个定义域的内点或者定义域的一个边界点。边界的每一条边都提供一个维数少1的最小值问题。计算F值的(s,t)点集和从这些值中选取的最小值包括如下的临界条件：

（1）$\nabla F(s,t)=\vec{0}$，其中，$(s,t)\in(s_{\min}, s_{\max})\times(t_{\min}, t_{\max})$

（2）$\partial F(s_{\min}, t)/\partial t=0$，其中，$t\in(t_{\min}, t_{\max})$

（3）$\partial F(s_{\max}, t)/\partial t=0$，其中，$t\in(t_{\min}, t_{\max})$

（4）$\partial F(s, t_{\min})/\partial s=0$，其中，$s\in(s_{\min}, s_{\max})$

（5）$\partial F(s, t_{\max})/\partial s=0$，其中，$s\in(s_{\min}, s_{\max})$

（6）$(s_{\min}, t_{\min}), (s_{\min}, t_{\max}), (s_{\max}, t_{\min}), (s_{\max}, t_{\max})$

求解该偏导数方程的复杂程度与$F(s,t)$的复杂程度直接相关。

2. 代数表示法

设包含两条曲线的平面分别为$\hat{n}_0\cdot X=c_0$和$\hat{n}_1\cdot Y=c_1$，其中X表示第一条曲线上的任意点，Y表示第二条曲线上的任意点。假设它们的法线向量都是单位长度的，而且它们不是同一平面。这也允许曲线位于平行的平面上。假设第一条曲线定义为包含它的平面和一个用多项式方程$P_0(X)=0$所隐式定义的曲面之间的交线。我们可以假定∇P_0与平面的法线$\hat{n}_0$不平行，这是一种合理的假设，因为只有曲面横切平面才能形成曲线。同样，假设第二条曲线定义为包含它的平面和一个用多项式方程$P_1(Y)=0$所隐式定义的曲面之间的相交，其中∇P_1与平面的法线$\hat{n}_1$不平行。对每一个点对（各位于一条曲线上）之间的距离平方为$\|X-Y\|^2$。两条曲线之间的距离可通过一对使$\|X-Y\|^2$在4个代数约束条件下取最小值的一对(X,Y)来获得。

使用A.9.3节讨论的拉格朗日乘子法可用来计算最小值。定义

$$G(X,Y,s_0,t_0,s_1,t_1)=\|X-Y\|^2+s_0\hat{n}_0\cdot(X-C_0)+t_0P_0(X)$$
$$+s_1\hat{n}_1\cdot(Y-C_1)+t_1P_1(Y)$$

我们知道$G:\mathbb{R}^{10}\to\mathbb{R}$。导数$\partial G/\partial X$表示$G$对于$X$的三个分量的第一阶偏导数所构成的三元组。导数$\partial G/\partial Y$表示$G$对于$Y$的三个分量的第一阶偏导数所构成的三元组。当$\partial G/\partial X=\vec{0}$，$\partial G/\partial Y=\vec{0}$，$\partial G/\partial s_0=0$，$\partial G/\partial t_0=0$，$\partial G/\partial s_1=0$且$\partial G/\partial t_1=0$时，函数$G$具有最小值。最后的4个导数方程只是重述了定义两条曲线的4个约束条件。X和Y的偏导数方程为

$$\frac{\partial G}{\partial X}=2(X-Y)+s_0\hat{n}_0+t_0\nabla P_0(X)=\vec{0} \quad 且$$

$$\frac{\partial G}{\partial Y}=2(Y-X)+s_1\hat{n}_1+t_1\nabla P_1(\vec{T})=\vec{0}$$

分别用 $\hat{n}_0\times\nabla P_0$ 和 $\hat{n}_1\times\nabla P_1$点乘上述导数方程，得到 $\hat{n}_0\times\nabla P_0(X)\cdot(X-Y)=0$ 和 $\hat{n}_1\times\nabla P_1(Y)\cdot(Y-X)=0$。将它们与 4 个约束条件组合起来可得到$X$和$Y$的 6 个未知分量的多项式方程

$$\hat{n}_0\times X=c_0$$

$$\hat{n}_1\times Y=c_1$$

$$P_0(X)=0$$

$$P_1(Y)=0$$

$$\hat{n}_0\times\nabla P_0(X)\cdot(X-Y)=0$$

$$\hat{n}_1\times\nabla P_1(Y)\cdot(Y-X)=0$$

与计算直线与平面曲线之间的距离一样，变量可从方程中直接消元，以得到一个关于一个变量的高次数多项式方程。求解该方程的根，从不同的临时多项式方程中构造剩余的 5 个分量这样我们已知所有可能的X和Y，然后用这些点来计算F的值。两条曲线之间的距离平方就是所有这些F值的最小值。

使用平面方程来消元两个变量可能得到较低次数的多项式方程。也就是说，当$i=0$和$i=1$时，$\{\hat{u}_i,\hat{v}_i,\hat{n}_i\}$是一个右手正交集，并且如果$C_i$是平面上的点，那么$X=C_0+u_0\hat{u}_0+v_0\hat{v}_0$且$Y=C_1+u_1\hat{u}_1+v_1\hat{v}_1$。将它们代入其余的 4 个约束条件，可得到 4 个关于 4 个未知数 u_0，v_0，u_1和 v_1的多项式方程。对这些方程应用消元方法，可得到比使用X和Y的 6 个分量时次数更低的多项式方程。

【实例】三维空间中圆与圆之间的距离。对于三维空间中的圆，上述讨论中提及的多项式 P_0 和 P_1 分别为$P_0(X)=\|X-C_0\|^2-r_0^2$ 和$P_1(Y)=\|Y-C_1\|^2-r_1^2$，其中C_i 为圆心，r_i 为半径。隐含的曲面是球面，它们与平面的交线都是圆。基于平面方程可得表达式$X=C_0+u_0\hat{u}_0+v_0\hat{v}_0$且$Y=C_1+u_1\hat{u}_1+v_1\hat{v}_1$。圆可用$u_i^2+v_i^2=r_i^2$来表示，其中$i=0,1$。由于$\nabla P_0(X)=2(X-C_0)=2r_0(u_0\hat{u}_0+v_0\hat{v}_0)$，因此法线与梯度的叉积为$\hat{n}_0\times\nabla P_0=2r_0(u_0\hat{v}_0-v_0\hat{u}_0)$。我们已经利用了$\{\hat{u}_0,\hat{v}_0,\hat{n}_0\}$是右手正交系这一事实。类似地，$\hat{n}_1\cdot\nabla P_1=2r_1(u_1\hat{v}_1-v_1\hat{u}_1)$。圆的方程和用拉格朗日乘子法求得的两个方程为

$$u_0^2+v_0^2=r_0^2$$

$$u_1^2+v_1^2=r_1^2$$

$$(u_0\hat{v}_0-v_0\hat{u}_0)\cdot(C_0-C_1-u_1\hat{u}_1-v_1\hat{v}_1)=0$$

$$(u_1\hat{v}_1-v_1\hat{u}_1)\cdot(C_1-C_0-u_0\hat{u}_0-v_0\hat{v}_0)=0$$

这是一个具有 4 个未知数u_0，v_0，u_1和v_1的二次多项式方程。

设$\hat{d}=C_0-C_1$，最后两个方程的形式为

$$u_0(a_0 + a_1u_1 + a_2v_1) + v_0(a_3 + a_4u_1 + a_5v_1) = 0$$

$$u_1(b_0 + b_1u_0 + b_2v_0) + v_1(b_3 + b_4u_0 + b_5v_0) = 0$$

其中

$$a_0 = \hat{v}_0 \cdot \hat{d},\ a_1 = -\hat{v}_0 \cdot \hat{u}_1,\ a_2 = -\hat{v}_0 \cdot \hat{v}_1,\ a_3 = -\hat{u}_0 \cdot \hat{d},\ a_4 = \hat{u}_0 \cdot \hat{u}_1,\ a_5 = \hat{u}_0 \cdot \hat{v}_1$$

$$b_0 = -\hat{v}_1 \cdot \hat{d},\ b_1 = -\hat{v}_1 \cdot \hat{u}_0,\ b_2 = -\hat{v}_1 \cdot \hat{v}_0,\ b_3 = \hat{u}_1 \cdot \hat{d},\ b_4 = \hat{u}_1 \cdot \hat{u}_0,\ b_5 = \hat{u}_1 \cdot \hat{v}_0$$

用矩阵形式来表示，有

$$\begin{bmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{bmatrix} \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} a_0 + a_1u_1 + a_2v_1 & a_3 + a_4u_1 + a_5v_1 \\ b_1u_1 + b_4v_1 & b_2u_1 + b_5v_1 \end{bmatrix} \begin{bmatrix} u_0 \\ v_0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 \\ -(b_0u_1 + b_3v_1) \end{bmatrix} = \begin{bmatrix} 0 \\ \lambda \end{bmatrix}$$

设M表示方程中 2×2 矩阵。方程两边乘以M的伴随矩阵，可得

$$\det(\mathrm{M}) \begin{bmatrix} u_0 \\ v_0 \end{bmatrix} = \begin{bmatrix} m_{11} & -m_{01} \\ -m_{10} & m_{00} \end{bmatrix} \begin{bmatrix} 0 \\ \lambda \end{bmatrix} = \begin{bmatrix} -m_{01}\lambda \\ m_{00}\lambda \end{bmatrix} \tag{10.17}$$

将各向量分量的平方相加，利用$u_0^2 + v_0^2 = r_0^2$，并与左边相减，可得

$$r_0^2 \left(m_{00}m_{11} - m_{01}m_{10}\right)^2 - \left(m_{00}^2 + m_{01}^2\right) \lambda^2 = 0 \tag{10.18}$$

这是一个关于u_1和v_1的二次多项式方程。

方程（10.18）可简化为一个次数为 8 的多项式方程，其根$v_1 \in [-1, 1]$是提供F的全局最小值的候选值。计算其行列式并利用$u_1^2 = r_1^2 - v_1^2$可得到$m_{00}m_{11} - m_{01}m_{10} = p_0(v_1) + u_1p_1(v_1)$，其中$p_0(z) = \sum_{i=0}^{2} p_{0i}z^i$和$p_1(z) = \sum_{i=0}^{1} p_{1i}z$。系数为

$$p_{00} = r_1^2(a_1b_2 - a_4b_1), \qquad p_{10} = a_0b_2 - a_3b_1,$$

$$p_{01} = a_0b_5 - a_3b_4, \qquad p_{11} = a_1b_5 - a_5b_1 + a_2b_1 - a_4b_4,$$

$$p_{02} = a_2b_5 - a_5b_4 + a_4b_1 - a_1b_2$$

类似地，$m_{00}^2 + m_{01}^2 = q_0(v_1) + u_1q_1(v_1)$，其中$q_0(z) = \sum_{i=0}^{2} q_{0i}z^i$和$q_1(z) = \sum_{i=0}^{1} q_{1i}z$。系数为

$$q_{00} = a_0^2 + a_3^2 + r_1^2(a_1^2 + a_4^2), \quad q_{10} = 2(a_0a_1 + a_3a_4),$$

$$q_{01} = 2(a_0a_2 + a_3a_5), \qquad q_{11} = 2(a_1a_2 + a_4a_5),$$

$$q_{02} = a_2^2 + a_5^2 - a_1^2 - a_4^2$$

最后，$\lambda^2 = r_0(v_1) + u_1r_1(v_1)$，其中$r_0(z) = \sum_{i=0}^{2} r_{0i}z^i$和$r_1(z) = \sum_{i=0}^{1} r_{1i}z$。系数为

$$r_{00} = r_1^2b_0^2, \quad r_{10} = 0,$$

$$r_{01} = 0, \qquad r_{11} = 2b_0b_3,$$

$$r_{02} = b_3^2 - b_0^2$$

替换方程（10.18）中的p，q和r，并利用$u_1^2 = 1 - v_1^2$，可得

$$0 = r_0^2[p_0(v_1) + u_1p_1(v_1)]^2 - [q_0(v_1) + u_1q_1(v_1)][r_0(v_1) + u_1r_1(v_1)] = g_0(v_1) + u_1g_1(v_1) \tag{10.19}$$

其中$g_0(z)=\sum_{i=0}^{4}g_{0i}z^i$和$g_1(z)=\sum_{i=0}^{3}g_{1i}z^i$。系数为

$$g_{00}=r_0^2(p_{00}^2+r_1^2p_{10}^2)-q_{00}r_{00}$$

$$g_{01}=2r_0^2(p_{00}p_{01}+r_1^2p_{10}p_{11})-q_{01}r_{00}-q_{10}r_1^2r_{11}$$

$$g_{02}=r_0^2(p_{01}^2+2p_{00}p_{02}-p_{10}^2+r_1^2p_{11}^2)-q_{02}r_{00}-q_{00}r_{02}-r_1^2q_{11}r_{11}$$

$$g_{03}=2r_0^2(p_{01}p_{02}-p_{10}p_{11})-q_{01}r_{02}+q_{10}r_{11}$$

$$g_{04}=r_0^2(p_{02}^2-p_{11}^2)-q_{02}r_{02}+q_{11}r_{11}$$

$$g_{10}=2r_0^2p_{00}p_{10}-q_{10}r_{00}$$

$$g_{11}=2r_0^2(p_{01}p_{10}+p_{00}p_{11})-q_{11}r_{00}-q_{00}r_{11}$$

$$g_{12}=2r_0^2(p_{02}p_{10}+p_{01}p_{11})-q_{10}r_{02}-q_{01}r_{11}$$

$$g_{13}=2r_0^2p_{02}p_{11}-q_{11}r_{02}-q_{02}r_{11}$$

通过计算$g_0=-u_1g_1$可以消元u_1，平方，并与左式相减可得$0=g_0^2-(r_1^2-v_1^2)g_1^2=h(v_1)$，其中$h(z)=\sum_{i=0}^{8}h_iz^i$。系数为

$$h_0=g_{00}^2-r_1^2g_{10}^2$$

$$h_1=2(g_{00}g_{01}-r_1^2g_{10}g_{11})$$

$$h_2=g_{01}^2+g_{10}^2+2g_{00}g_{02}-r_1^2(g_{11}^2+2g_{10}g_{12})$$

$$h_3=2(g_{01}g_{02}+g_{00}g_{03}+g_{10}g_{11})-2r_1^2(g_{11}g_{12}+g_{10}g_{13})$$

$$h_4=g_{02}^2+g_{11}^2+2(g_{01}g_{03}+g_{00}g_{04}+g_{10}g_{12})-r_1^2(g_{12}^2+2g_{11}g_{13})$$

$$h_5=2(g_{02}g_{03}+g_{01}g_{04}+g_{11}g_{12}+g_{10}g_{13}-r_1^2g_{12}g_{13})$$

$$h_6=g_{03}^2+g_{12}^2-r_1^2g_{13}^2+2(g_{02}g_{04}+g_{11}g_{13})$$

$$h_7=2(g_{03}g_{04}+g_{12}g_{13})$$

$$h_8=g_{04}^2+g_{13}^2$$

为了找到距离平方的最小值，计算$h(v_1)=0$的所有实数根。对每一个根$\bar{v}_1\in[-1,1]$，计算$\bar{u}_1=\pm\sqrt{1-\bar{v}_1^2}$并选取其中一个或两个满足方程（10.19）的根。对每一对$(\bar{u}_1,\bar{v}_1)$，求解方程（10.17）的根$(\bar{u}_0,\bar{v}_0)$。其中主要的数值问题就是det(M)将如何接近于零。

最后，计算距离平方$\|X-Y\|^2$，其中$X=C_0+\bar{u}_0\hat{u}_0+\bar{v}_0\hat{v}_0$且$Y=C_1+\bar{u}_1\hat{u}_1+\bar{v}_1\hat{v}_1$。这些距离平方值中的最小值就是两个圆之间的距离平方值。■

10.13.4　曲面上的测地距离

可以在关于曲线和曲面的微分的几何参考书中找到下面所讨论的内容。**Kay**（1988）编写的参考书是其中特别容易理解的一本。给定曲面上的两个点，我们要计算这两个点之

间沿着曲面的最短距离。连接这两个点的最短距离路径叫做测地曲线，该曲线的弧长叫做这两点之间的测地距离。对于平面上的两个点，最短路径就是连接这两个点的线段。可是，在一个曲面上，最短路径并不一定是惟一的。例如，球面上相对着的两个点具有无数的最短路径连接着它们，每一条路径都是球面上的一个大半圆。

这里讨论的建立测地曲线的方法是基于松弛的方法。由于数学细节太冗长，所以这里仅提供这种方法的基本思想。位于曲面上的连接两个点的初始曲线允许随着时间而发生变化。这种变化以热流模型为基础，并且在文献中以欧几里得曲线收缩为主题进行讨论已得到广泛的研究。这种思想也应用于其他的许多领域，特别是计算机视觉和图像处理领域（Haar Romeny 1994）。变化的曲线表示为$X(s,t)$，其中s是弧长参数，t是变化的时间。曲线的端点总是两个输入点，即P和Q。在平面上，这种思想允许曲线根据线性热量方程$\vec{X}_t=\vec{X}_{ss}$（其中$\vec{X}_t$是X对t的第一阶偏导数，$\vec{X}_{ss}$是X对s的第二阶偏导数。）而变化。虽然X是一个点数量，其偏导数却是向量，因此用向量标记来表示。当t趋近于无穷大时，曲线的极限将成为连接P和Q的线段。任何连接着两点的初始曲线$X(s,0)=C(s)$都可以看成是从其自然状态“伸展”而成的曲线。随着时间的增加，曲线允许“松弛”而变回自然状态，在这种情形中，就是连接这两个点的线段。

对于一个曲面，这种变化更加复杂一点：

$$
\begin{aligned}
&\vec{X}_t=\vec{X}_{ss}-(\vec{X}_{ss}\cdot\hat{n})\hat{n},\quad t>0\\
&X(s,0)=C(s),\\
&X(0,t)=P,\quad X(L(t),t)=Q
\end{aligned}
\tag{10.20}
$$

向量$\hat{n}(s,t)$是位于曲面上相关点$X(s,t)$的曲面法线。要求变化的曲线是位于曲面上的。任何点$X(s,t)$都仅能与曲线相切地移动。运动由时间偏导数$\vec{X}_t$来确定，因此$\vec{X}_t$必须是曲面的一个切线向量。变化方程的右手边有一个扩散项$\vec{X}_{ss}$，但是与法线向量相关的修正项只是将$\vec{X}_{ss}$的贡献在法线方向上投影掉，只留下切线分量。连接两个点的初始曲线是$C(s)$。$X(s,t)$的长度表示为$L(t)$。边界条件是限制曲线$X(s,t)$的端点必须是点P和Q的两个约束条件。随时间变化的边界条件$X(L(t),t)=Q$使得这一问题变得更加复杂。一般的关于偏微分方程的教科书仅仅讨论边界条件不随时间变化的问题。

求解变化方程（即方程（10.20））的数值方法使用s偏导数的中心差分近似，以及t偏导数的向前差分近似。即

$$
\vec{X}_{ss}(s,t)\doteq\frac{(X(s+h,t)-X(s,t))+(X(s-h,t)-X(s,t))}{h^2}
$$

以及

$$
\vec{X}_t(s,t)\doteq\frac{X(s,t+k)-X(s,t)}{k}
$$

如果已知$X(s,t)$并且曲面被隐式定义为$F(X)=0$，那么该点的曲面法线可隐式表示为$\hat{n}(s,t)=\nabla F(X(s,t))/\|\nabla F(X(s,t))\|$。如果曲面定义为参数形式$X(u,v)$，那么曲面法线为$\hat{n}(u,v)=\vec{X}_u\times\vec{X}_v/\|\vec{X}_u\times\vec{X}_v\|$。然而，变化方程并不知道曲面的参数$u$和$v$，因此必须用其他的方法来计算法线。只要偏导数$\vec{X}_t$和$\vec{X}_s$不平行（曲线在变化时不仅仅沿着切线方向伸展），那么法线向量可估算为$\hat{n}(s,t)\doteq\vec{X}_s(s,t)\times\vec{X}_t(s,t)/\|\vec{X}_s(s,t)\times\vec{X}_t(s,t)\|$。在变化方程中

代入这些近似值，可得

$$X(s,t+k)=X(s,t)+\frac{k}{h^2}\left(I-\hat{n}\hat{n}^{\mathrm{T}}\right)$$

$$((X(s+h,t)-X(s,t))+(X(s-h,t)-X(s,t))) \qquad (10.21)$$

选取一条初始曲线$C(s)$，以及曲线上的一组等距点s_i，$0\leqslant i\leqslant M$。选取的点的数目随应用程序而定，但是一般地，数目越多，得到的测地曲线就越精确。选取时间步长$k>0$和空间步长$h>0$。比值k/h^2应该足够小，以保证数值的稳定。该值应该多小取决于应用曲线，而且要用一般的技术来确定其稳定性。如果对所有的i有$X(s_i,0)=C(s_i)$，那么在时间$t=k$处的曲线样本将在方程（10.21）的左手边$X(s_i,k)$计算。数值误差能导致$X(s_i,k)$偏离曲面。如果曲面由$F(X)=0$隐式定义，那么定义方程潜在地可能用于纠正$X(s_i,k)$，以将其校正回曲面。如果曲面用参数形式来定义，纠正就更困难一些，因为并不能直接知道如何选择参数u和v，才能使$X(u,v)$最接近于曲面上的$X(s_i,k)$。

方程（10.21）是递归的，直到满足某些停止条件。存在许多的选择，包括：在时间t和$t+k$之间测量所有样本点之间的总的变化，当变化足够小的时候就停止，或者计算连接样本点的折线的弧长，当在时间t和$t+k$之间的弧长变化足够小的时候就停止。在每一种情形中，最后的折线弧长都作为测地距离的近似值。

第 11 章　三维相交

本章包含了计算三维几何对象相交的信息。最简单的对象组合是其中的一个对象为线形对象（直线、射线、线段）的情形。前面的 4 节将介绍这类组合。接下来的 4 节将介绍平面对象（平面、三角形、多面体）与其他对象（平面对象之间、与多面体、二次曲面以及多项式曲面）的相交。其中的两节将介绍二次曲面与另一个二次曲面的相交，以及多项式曲面与另一个多项式曲面的相交。还有一节介绍了轴分离的方法，这是一种非常有效的处理凸形对象相交的技术。最后一节将介绍相交问题的杂项问题。

11.1　线形对象与平面对象的相交

本节涵盖了计算三维空间中线形对象与平面对象的相交问题。线形元素包括射线、线段和直线。定义这样的几何实体有许多不同的方法（参看 9.1 节）。为了便于本节的讨论，我们使用与坐标无关的参数表示，即用一个基点P和一个方向$\vec{d}$来定义一条线形对象$\mathcal{L}$：

$$\mathcal{L}(t) = P + t\vec{d} \tag{11.1}$$

经常使用规整化向量来定义射线$\mathcal{R}$

$$\mathcal{R}(t) = P + t\hat{d}, \qquad 0 \leqslant t \leqslant \infty \tag{11.2}$$

但是并不一定使用规整化向量来定义直线$\mathcal{L}$

$$\mathcal{L}(t) = P + t\vec{d}, \qquad -\infty \leqslant t \leqslant \infty \tag{11.3}$$

我们假设用一对点$\{P_0, P_1\}$来表示线段$\mathcal{S}$。通过将线段转化为如下的射线形式，我们就能使用与处理射线 / 平面对象之间的相交算法相似的算法来处理线段：

$$\mathcal{S}(t) = P_0 + t(P_1 - P_0)$$

也就是说，我们的方向向量$\vec{d}$由两个不同的点所定义。注意，一般地，$\|\vec{d}\| \neq 1$。然而，方向向量不但不需要是规整化的，而且也不必要规整化：因为$P_1 = P_0 + \vec{d}$，所以如果我们计算这条“射线”与平面对象的相交，那么当且仅当$0 \leqslant t \leqslant 1$时，交点在线段上。

11.1.1　线形对象与平面的相交

在本节中，我们讨论线形对象与平面的相交问题。一个平面$\mathcal{P}$定义为$[a \quad b \quad c \quad d]$：

$$ax + by + cz + d = 0 \tag{11.4}$$

其中$a^2 + b^2 + c^2 = 1$。其中一个向量$\hat{n} = [a \quad b \quad c]$表示平面的法线，其中$|d|$表示从平面到原点$[0 \quad 0 \quad 0]$的最小距离。

如图 11.1 所示，线形对象$\mathcal{L}$和$\mathcal{P}$之间的相交（如果存在的话）位于点$Q = P + t\vec{d}$。由于Q是$\mathcal{P}$上的一个点，因此它必定也满足方程 11.4。

我们只需将方程（11.1）代入方程（11.4）：

$$a\left(P_x + d_x t\right) + b\left(P_y + d_y t\right) + c\left(P_z + d_z t\right) + d = 0$$

并求解其中的参数t：

$$t = \frac{-\left(aP_x + bP_y + cP_z + d\right)}{ad_x + bd_y + cd_z}$$

将上述方程中的式子看成是向量运算是非常有用的：

$$t = \frac{-\left(\hat{n} \cdot P + d\right)}{\hat{n} \cdot \vec{d}}$$

注意，分母$\hat{n} \cdot \vec{d}$表示平面的法线方向与射线方向的点积。如果该值等于 0，那么射线与平面平行。如果射线在平面内，那么存在无数的交点；如果射线不在平面内，那么不存在交点。由于浮点数只能近似表示实数及实数之间的运算，直线与平面很少是完全平行的，因此，应该将点积值与某些小的数值ϵ进行比较。ϵ的值取决于变量的精度和与应用程序相关的要素。先计算该值将使我们能快速地排除这种情形。

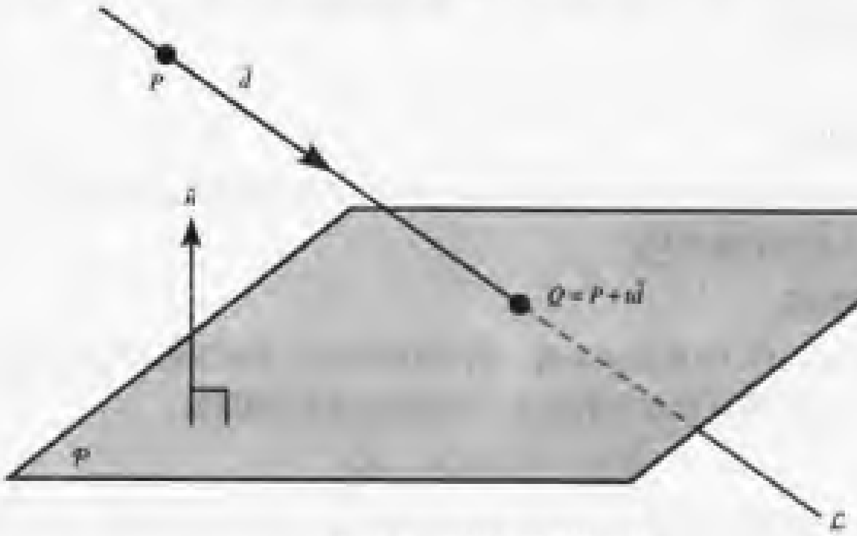


图 11.1 直线与平面的相交

求得分子后，我们就能进行除法以求得t。计算交点的坐标要求将我们求得的t值代入方程（11.2）：

$$Q = P + t\vec{d}$$

计算直线一平面相交的伪码如下

```
boolean LineIntersectPlane(
    Line3D   line,
    Plane    plane,
    float&   t,
    Point3D& intersection)
{
```

```
    // Check for (near) parallel line and plane
    denominator = Dot(line.direction, plane.normal)
    if (Abs(denominator) < epsilon) {
        // Check if line lies in the plane or not.
        // We do this, somewhat arbitrarily, by checking if
        // the origin of the line is in the plane. If it is,
        // set the parameter of the intersection to be 0. An
        // application may wish to handle this case differently...
        if (Abs(line.origin.x * plane.a + line.origin.y * plane.b +
                line.origin.z * plane.c + plane.d) < epsilon) {
            t = 0;
            return (true);
        } else {
            return false;
        }
    }

    // Nonparallel, so compute intersection
    t = -(plane.a * line.origin.x + plane.b * line.origin.y +
          plane.c * line.origin.z + plane.d);
    t = t / denominator;
    intersection = line.origin + t * line.direction;
    return true
}
```

1. 射线－平面相交

定义射线，只需要求 $t \geqslant 0$ ，因此我们只需检查用直线相交函数求得的t值是大于 0 还是等于 0，并据此认定或否定相交。

2. 线段－平面相交

我们假设用一对点$\{P_0, P_1\}$来表示线段。通过将线段转化为如下的射线形式，我们就能使用与处理射线 / 平面对象之间相交的算法一样的算法来处理线段：

$$\mathcal{R}(t) = P_0 + t(P_1 - P_0)$$

线段定义为$0 \leqslant t \leqslant 1$，因此我们只需将计算得到的$t$值与该范围比较，并据此认定或否定相交。

11.1.2　线形对象与三角形的相交

在本节中，我们将讨论射线、直线和线段与三角形的相交问题。在其中一个子节中，我们将讨论更一般的线形元素与多边形的相交问题。当然也可以将直线 / 三角形的相交问题看成是三个顶点的多边形的特例，但是我们能利用重心坐标，并据此得到一种更直接有效的方法。

一种方法是将直线与包含三角形的平面相交，再确定交点是否位于三角形之内。通过将三角形的顶点和交点投影到一个轴对齐平面（选取使投影所得的三角形面积最大的平面），然后通过二维空间的点是否在三角形内的测试就可以确定这种包含关系（参看 Haines 1994 和 13.3.1 节）。然而，这样的方法要求在每次进行相交测试时都计算三角形的法线或

者存储这些法线（并保证在它改变后要重新计算）。

另一种方法是使用 Möller 和 Trumbore（1997）提出的方法。我们也将线形对象定义为一个基点和一个方向向量（方程（11.1））。可以简单地将一个三角形定义为一组有序顶点 $\{V_0, V_1\ V_2\}$（如图 11.2 所示）。

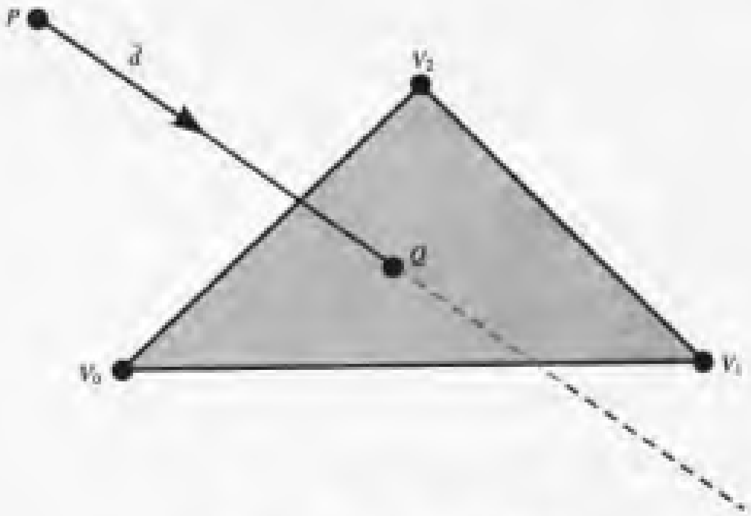


图 11.2　直线与三角形的相交

复习一下，三角形内的任何点都可以用它相对于三角形的顶点的位置来定义：

$$Q_{u,v,w} = wV_0 + uV_1 + vV_2 \tag{11.5}$$

其中 $u + v + w = 1$。三元组 (u, v, w) 称为 Q 的重心坐标（barycentric coordinates）。由于 $w = 1 - (u + v)$，因此我们经常仅使用 (u, v) 来表示重心坐标（参看 3.5 节）。

与计算线形对象与平面的相交一样，只需将方程（11.2）代入方程（11.5）中，就能计算线形对象－三角形的相交：

$$P + t\hat{d} = (1 - (u + v))V_0 + uV_1 + vV_2$$

该式可以扩展为

$$\begin{bmatrix} -\hat{d} & V_1 - V_0 & V_2 - V_0 \end{bmatrix} \begin{bmatrix} t \\ u \\ v \end{bmatrix} = \begin{bmatrix} P - V_0 \end{bmatrix}$$

我们知道，每一个这样的变量都具有向量值，可以看出，这是一个由三个具有三个未知数的方程所组成的线性系统。可以用许多的方法来求解这类系统，但是我们在此仅选用克拉默法则（参见 2.7.4 节和 A.1 节）。

基于克拉默法则，我们有

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{|-\hat{d} \quad V_1 - V_0 \quad V_2 - V_0|} \begin{bmatrix} |P - V_0 \quad V_1 - V_0 \quad V_2 - V_0| \\ |-\hat{d} \quad P - V_0 \quad V_2 - V_0| \\ |-\hat{d} \quad V_1 - V_0 \quad P - V_0| \end{bmatrix}$$

$$= \frac{1}{(\hat{d} \times (V_2 - V_0)) \cdot (V_1 - V_0)} \begin{bmatrix} ((P - V_0) \times (V_1 - V_0)) \cdot (V_2 - V_0) \\ (\hat{d} \times (V_2 - V_0)) \cdot (P - V_0) \\ ((P - V_0) \times (V_1 - V_0)) \cdot \hat{d} \end{bmatrix}$$

可以进行最后一步的改写是因为有

$$\begin{aligned}|\vec{u} \quad \vec{v} \quad \vec{w}| &= -(\vec{u} \times \vec{w}) \cdot \vec{v} \\ &= -(\vec{w} \times \vec{v}) \cdot \vec{u}\end{aligned}$$

以及进行了常用的运算$\hat{d} \times (V_2 - V_0)$和$(P - V_0) \times (V_1 - V_0)$。

一旦求得了t，u和v，我们就能通过考察它们的值来确定交点是否位于三角形内（而不是位于多边形所在平面的其他地方）：如果$0 \leqslant u \leqslant 1$，$0 \leqslant v \leqslant 1$并且$u + v \leqslant 1$，那么相交位于三角形内；否则，它位于多边形平面上，但位于三角形外。

进行上述计算的伪码为

```
bool LineTriangleIntersect(
    Triangle3D tri,
    Line3D     line,
    Isect&     info,
    float      epsilon,
    Point3D&   intersection)
{
    // Does not cull back-facing triangles.
    Vector3D e1, e2, p, s, q;
    float t, u, v, tmp;
    e1 = tri.v1 - tri.v0;
    e2 = tri.v2 - tri.v0;
    p = Cross(line.direction, e2);
    tmp = Dot(p, e1);

    if (tmp > -epsilon && tmp < epsilon) {
        return false;
    }

    tmp = 1.0 / tmp;
    s = line.origin - tri.v0;

    u = tmp * Dot(s, p);
    if (u < 0.0 || u > 1.0) {
        return false;
    }

    q = Cross(s, e1);
    v = tmp * Dot(d, q);

    if (v < 0.0 || v > 1.0) {
        return false;
    }

    t = tmp * Dot(e2, q);

    info.u = u;
    info.v = v;
    info.t = t;
```

```
        intersection = line.origin + t * line.direction;
        return true;
    }
```

1. 射线—三角形相交

定义射线，只需要求$t \geq 0$，因此我们只需检查用直线相交函数求得的t值是大于 0 还是等于 0，并据此认定或否定相交。

2. 线段—三角形相交

我们假设用一对点$\{P_0, P_1\}$来表示线段。通过将线段转化为射线形式，我们就能使用与处理射线／三角形之间相交一样的算法来处理线段。线段定义为$0 \leq t \leq 1$，因此我们只需将计算得到的t值与该范围比较，并据此认定或否定相交。

11.1.3 线形对象与多边形的相交

我们能够确定交点的重心坐标，这有助于计算线形对象与三角形之间的交点。可以保证（在浮点误差范围内）交点位于三角形所在的平面上。可是这种技巧不能直接应用于一般的多边形情形。对于自身不相交的多边形，在理论上可以将它们分成三角形，然后对每一个三角形应用线形对象—三角形的相交算法，但是这样做的效率不高。

在下面的几节中，多边形被假设为在浮点误差范围内的面状对象，其自身不相交、封闭并且包含一条轮廓线。多边形表示为一组n个顶点：$\{V_0, V_1, \ldots, V_{n-1}\}$。多边形所在的平面用其顶点来隐式表示，并表示为一般的形式（即表示为一条法线和距原点的距离）：$ax + by + cz + d = 0$，其中$a^2 + b^2 + c^2 = 1$。

因为我们（一般）不能利用用于三角形情形的“重心坐标技巧”，计算线形对象—多边形之间的相交需要如下几个步骤：

（1）计算多边形所在的平面的方程。选取任一个顶点作为平面上的一个点，然后利用该顶点与其邻点构成的向量的叉积来计算平面的法线，这样就能得出平面的方程。然而，一般的多边形并不是严格面状的，因此应该使用更健壮的方法，比如 Newell 方法（Tampieri 1992）或 A.7.4 节中介绍的超面适应法。

（2）计算线形对象与平面之间的相交（参见 11.1.1 节）。

（3）如果线形对象与多边形所在的平面相交，那么确定交点是否位于多边形的边界内。

最后一步相当于检查射线／三角形相交的重心坐标，但是，对于多边形，我们必须使用一个“技巧”。这种技巧就是：将多边形的顶点和交点Q投影到由一个本地坐标系（XY，YZ或XZ平面）所定义的平面上，然后确定投影所得的交点Q'是否位于投影所得的多边形$\{V_0', V_1', \ldots, V_{n-1}'\}$之内（如图 11.3 所示）。

由于我们希望的投影为正交投影，因此投影所在的步骤中包括选择一个被忽略的坐标，并使用其他的两个坐标作为二维空间中的(x, y)坐标。被忽略的一个坐标应该是相对多边形的所有顶点变化最小的那个，即我们计算一个有界箱，被排除的坐标应该是对应于该箱子的最短的边。通过这样的处理，由于投影产生的数值误差将被最小化，特别是当多边形非常接近于与其中一个正交平面共面时。

因此，在最后一步中，我们只需解决二维空间中的点在多边形内的测试问题，有许多

的算法可以用来求解这类问题（参见 13.3 节）。

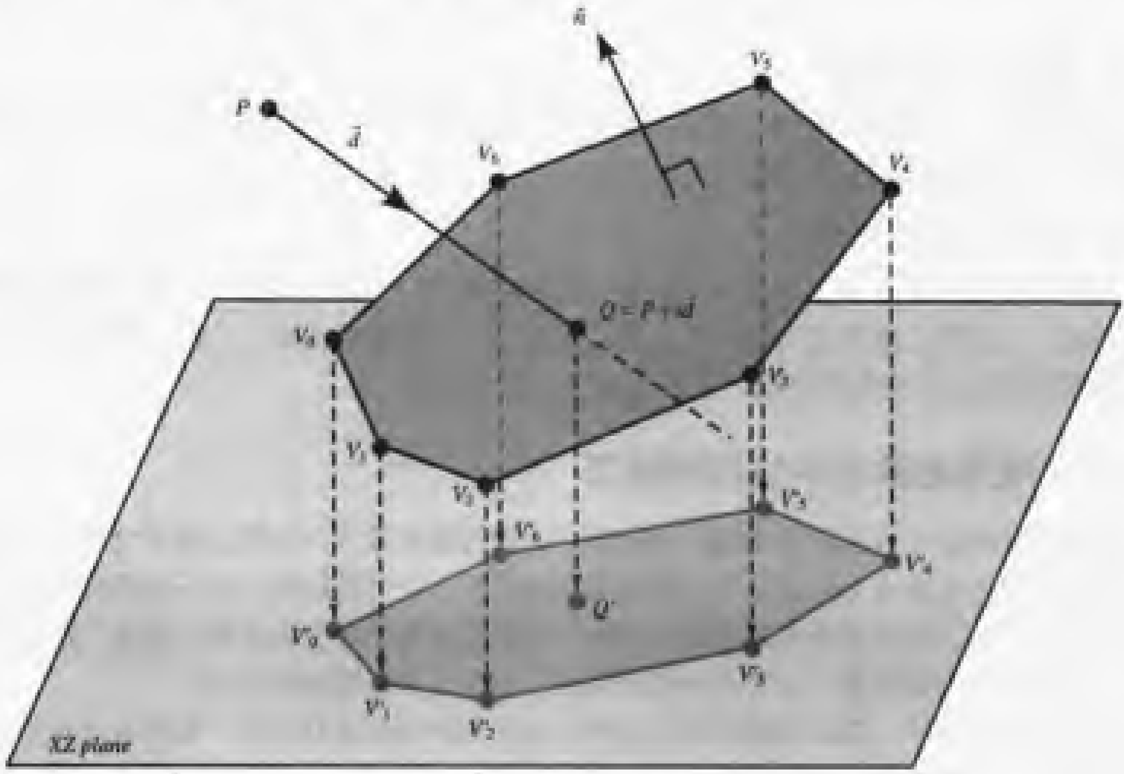


图 11.3　射线与多边形的相交

我们应该注意，多边形可以被定义为一个平面与一组半空间的相交，通过将每一对顶点 $\{V_i, V_{i+1}\}$ 看成是位于一个与多边形所在的平面垂直的平面上的两个点来定义这些半空间。然后，我们就能确定直线与多边形所在的平面的交点是否位于这些半空间的同一面。可以利用与另一种方法所使用的判断二维空间中点是否在多边形内的相同算法。这样又产生了另一个问题，即"既然我们使用（基本上）一样的方法，那么为什么要将点投影？"答案是效率——在（投影后的）二维空间上进行处理显然要快得多。

这种处理的伪码为

```
bool LinePolygonIntersection(
    Polygon3D poly,
    Line3D    line,
    float&    t,
    Point3D&  intersection)
{
    // lcp direction is assumed to be normalized
    // Also assumes polygon is planar
    Vector3D N, p, e1, e2;
    float numer, denom;
    e1 = poly.vertexPosition(1) - poly.vertexPosition(0);
    e2 = poly.vertexPosition(2) - poly.vertexPosition(1);
    N = Cross(e1, e2);
    N /= N.length();
    p = poly.vertexPosition(0);
```

```
        denom = Dot(line.direction, N);

        if (denom < 0) {
            numer = Dot(N, p - line.origin);
            t = numer / denom;
            if (t < 0) {
                return false;
            }
            p = line.origin + t * r.d;
            int projectionIndex = MaxAbsComponent(N);

            Point2D* 2dPoints;
            Point2D p2d;
            2dPoints = new Point2D[poly.numVertices];

            //  Project Points into a 2D plane
            //  by removing the coordinate that
            //  was the fabs maximum in the normal
            //  return them in array 2dPoints.
            Project2D(poly.VertexArray, projectionIndex, 2dPoints,
                      poly.numVertices);
            Project2D(p, projectionIndex, p2d, 1);

            // Choose your method of winding test
            // Sign of dotProducts etc...
            if (PointIn2DPolygon(p2d, 2dPoints)) {
                delete [] 2dPoints;
                intersection = line.origin + t * line.direction;
                return true;
            } else {
                delete [] 2dPoints;
                return false;
            }
        } else {
            // Back facing
            return false;
        }
    }
```

1. 射线—多边形相交

定义射线，只需要求$t \geqslant 0$，因此我们只需检查用直线相交函数求得的t值是否为非负，并据此认定或否定相交。

2. 线段—多边形相交

我们假设用一对点$\{P_0, P_1\}$来定义线段。通过将线段转化为射线形式，我们就能使用与处理射线／多边形之间的相交的算法一样的算法来处理线段。线段定义为$0 \leqslant t \leqslant 1$，因此我们只需检查计算得到的$t$是否位于该范围内，并据此认定或否定相交。

11.1.4　线形对象与圆盘的相交

在本节中，我们将讨论线形对象与圆盘的相交问题（如图 11.4 所示）线形对象定义为如下的一般形式：

$$\mathcal{L}(t) = P + t\vec{d}$$

圆盘定义为与三维空间中的圆相同的形式（参见 9.2.3 节）：

$$P = C + r\hat{w}_\theta$$

其中

$$\hat{w}_\theta = \cos\theta\hat{u} + \sin\theta\hat{v}$$

简单地说，圆盘就是三维空间中的圆及其周界所包围的面状区域：如果一条直线穿过圆的“内部”，那么直线与圆不相交，但是，如果一条直线穿过圆盘的“内部”，那么直线与圆盘相交。换一种方法，我们可以将圆盘定义为一个中点 C，一个平面法线 $\hat{n}$ 和一个半径 r。然而，这样却失去了任何的（相对于圆的“轴”的交点的）参数信息——其与应用程序的相关性是不确定的。

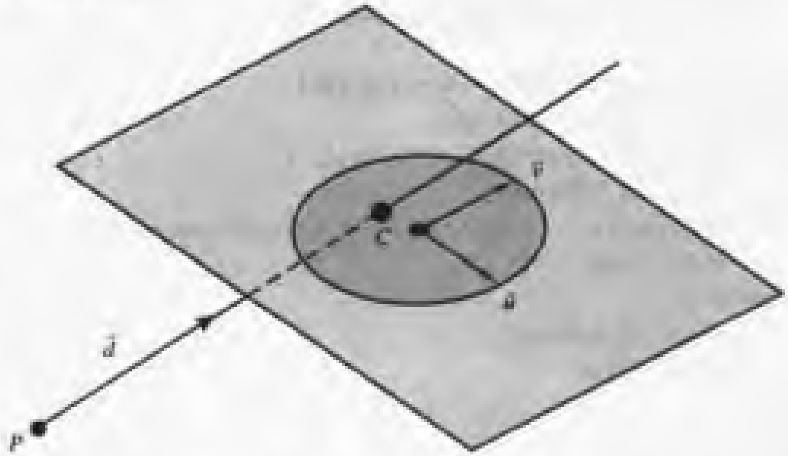


图 11.4　线形对象与圆盘的相交

简单地说，这种算法就是先求线形对象与圆盘所在平面的相交，然后计算交点与圆盘的中点之间的距离平方，并将其与半径的平方进行比较。如果线形对象位于圆盘所在的平面上，那么应用程序是否希望考虑相交问题是不确定的。如果在这种情形中，希望考虑相交问题，可以使用二维空间中的线形对象－圆的相交算法的三维空间扩展；如果应用程序仅仅关心是否相交，那么只需将线性对象与圆之间的距离（平方）与半径（平方）进行比较。

伪码为

```
bool LineIntersectDisk(Line3D line, Disk3D disk, Point3D p)
{
    Plane3D plane;
    plane.normal = disk.normal;
    plane.p = disk.center;
    float t;
```

```
    Point3D intersection

    if (!LinePlaneIntersection(plane, line, t, p)) {
        return false;
    }

    if (DistanceSquared(p, disk.center) <= disk.radius * disk.radius) {
        return true;
    } else {
        return false;
    }
}
```

1. 射线—圆盘相交

定义射线，只需要求 $t \geq 0$，因此我们只需检查用直线相交函数求得的 t 值是否为非负，并据此认定或否定相交。

2. 线段—圆盘相交

我们假设用一对点 $\{P_0, P_1\}$ 来表示线段。通过将线段转化为射线形式，我们就能使用与处理射线 / 圆盘之间的相交的算法一样的算法来处理线段。线段定义为 $0 \leq t \leq 1$，因此我们只需检查计算得到的 t 是否位于该范围内，并据此认定或否定相交。

11.2 线形对象与多面体的相交

本节讨论线形对象与用多边形网格表示的多面体的相交问题。线形对象，即射线、直线和线段，用一个基点和一个向量来定义

$$\mathcal{L}(t) = P + t\vec{d}$$

在这种情形中，线段定义为两个点 P_0 和 P_1，我们设 $\vec{d} = P_1 - P_0$。多面体用 9.3 节中描述的定义来定义。为了方便本节的讨论，多边形网格就是多面体，并不要求是闭合的。图 11.5 显示了一条射线与一个八面体的相交，而图 11.6 显示了一条线段与一个三角形网格的相交。注意，并不要求多面体是正多面体，也并不要求多边形网格的所有面都是三角形的。

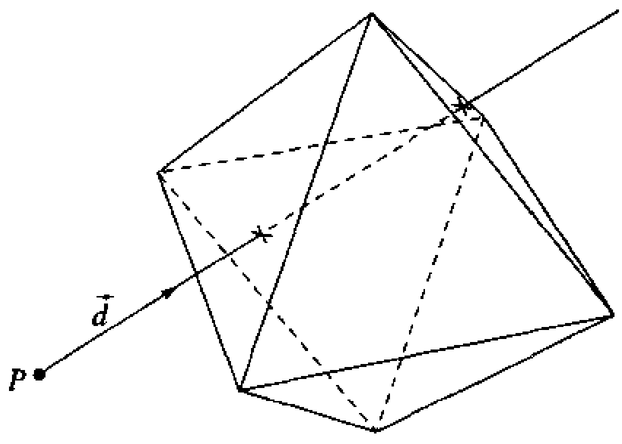


图 11.5 射线与多面体（八面体）的相交

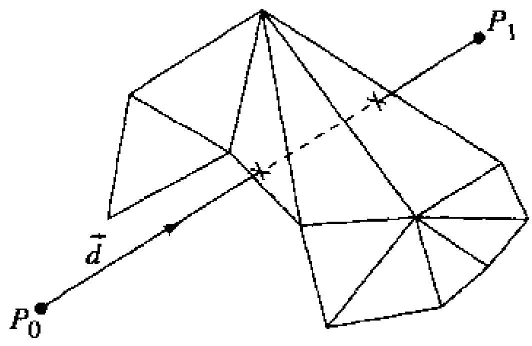


图 11.6　线段与多边形（三角形）网格的相交

由于多面体或多边形网格就是一组遵守特定的与共享的边和顶点相关的规则的多边形集合，因此最简单的方法就是测试每一个面的相交结果（参见 11.1.3 节）。然而，对于具有大量的面的多面体或多边形网格，这种初级的方法是非常低效的，因为需要花费大量的时间来计算并不与线形对象相交的面的相交。对于这种情形，应用程序应该采用一种空间分区方法，比如八叉树或二叉空间分区树（参见 Foley 等 1996，或本书第 13 章）。如果涉及大量的面，特别是多面体或多边形网格将相交多次（如在射线跟踪的例子中）时，建立这样的空间分区模型是非常值得的。

Eric Haines（1991）描述了一种以 Roth（1981）和 Kay 及 Kajiya（1986）中的思想为基础的算法，对于凸多面体，这种算法比初级方法要快得多。这种方法的一个优势是，它计算线形对象与包含每一个面的平面的相交（这相对要省时一些），而不是计算线形对象与多边形面自身的相交（这是非常费时的）。线形对象定义为

$$\mathcal{L}(t) = \mathcal{P} + t\hat{d}$$

而面所在的平面定义为

$$ax + by + cz + d = 0$$

给定上述定义，线形对象的原点$\mathcal{P}$与线形对象与多面体的面所在平面的交点之间的距离为

$$t_0 = \frac{-(\vec{n} \cdot \mathcal{P} + d)}{\vec{n} \cdot \hat{d}} \tag{11.6}$$

其中$\vec{n} = [a \quad b \quad c]$是平面的法线。如果方程（11.6）中的分母为（或接近于）零，那么线形对象与平面平行。在这种情形中，分子的符号说明线形对象的原点位于平面的哪一面。而分母的符号说明线形对象与平面相交于平面的正面还是背面：如果为正，则相交于平面的背面（用增加的 t 来表示），如果符号为负则相反。

隐藏在 Haines 算法后面的思想是：多面体所定义的体积可以理解为用多面体的面所在的平面所定义的半空间之间的逻辑相交。如果我们考虑一条线形对象，它与多面体的相交是其中的一个部分，这部分全部包含于半空间内。线形对象与每一个面所在的平面的相交将线形对象划分为两个区域，其中一个位于该平面所定义的半空间的“外面”，另一个位于该半空间的“里面”。从这些事实中，我们可以得出如下结论，即线形对象与整个多面体的相交部分，就是该部分（线形对象）位于每一个面所在的平面所定义的半空间之间的逻辑相交。如图 11.7 所示（为了清晰，显示为二维）。

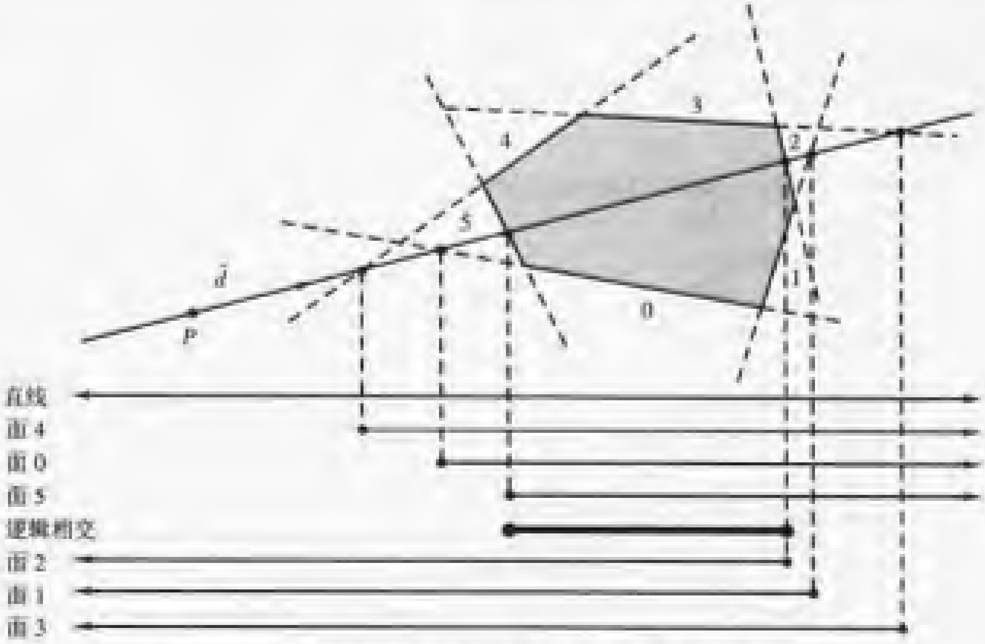


图 11.7 定义直线与多面体相交的半线的逻辑相交

注意，我们并没有按边的索引递增的次序来列出半线，而是按相交距离的增加次序；这样可以使它们的逻辑相交的性质更加明显。与多面体相交的任何直线都将先与一个或多个“前”面相交（如果认为直线“起始”于 $t=-\infty$），然后才与一些“背”面相交。逻辑相交被最后（最远的）一个“前”面和第一个（最近的）“背”面所包围。

注意，如果直线与多面体不相交，如图 11.8 所示，那么直线将与一个“背”面所在的平面相交之后，才与一个“前”面所在的平面相交，因此逻辑相交不存在。

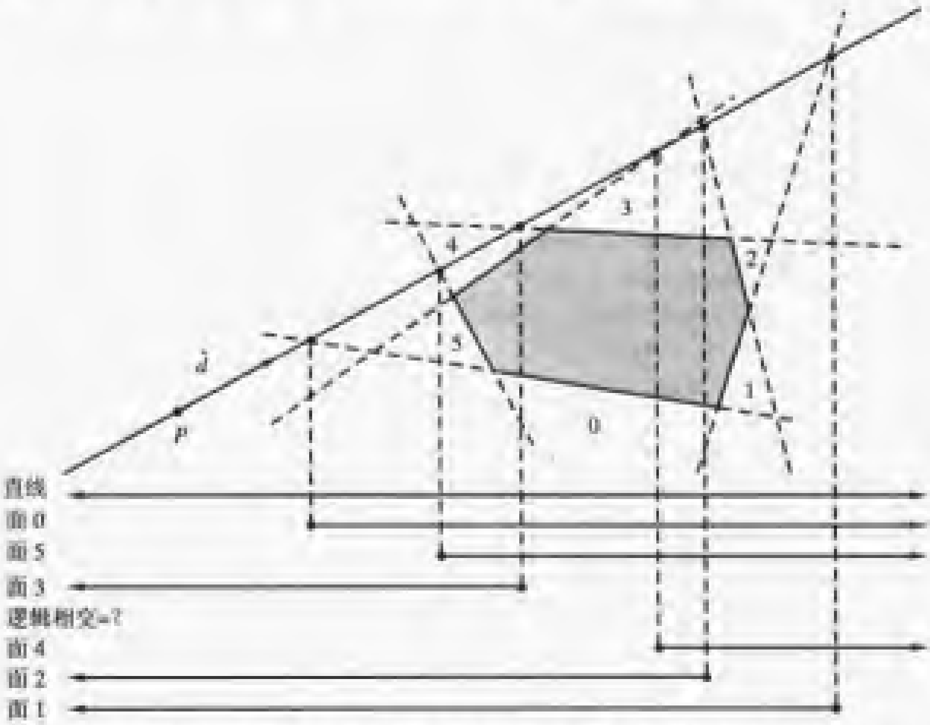


图 11.8 如果直线与多面体不相交，则不存在半线的逻辑相交

根据上述分析可以得到一个简单的算法。惟一还需要注意的一点是，如果我们找到与该直线平行的一个面所在的平面，并且发现直线指向平面的外面，那么就可以提前退出算法，因此在这种情形中，直线与多面体并不会相交。

伪码如下：

```
boolean LinePolyhedronIntersection(
    Line       line,
    Polyhedron phd,
    float&     tNear,
    float&     tFar)
{
    tNear = -MAXFLOAT;
    tFar  =  MAXFLOAT;

    foreach face F in polyhedron {
        normal = { F.a, F.b, F.c };
        denominator = Dot(normal, line.direction);
        numerator   = Dot(normal, line.origin) + F.d;
        if (denominator  < epsilon) {
            //
            //  Face F is parallel to the line. Check
            //  if line is outside the half-space
            //  defined by the plane
            //
            if (numerator > 0) {
                //
                //  Line is outside face and therefore
                //  outside the polyhedron.
                //
                return false;
            }
        } else {
            //
            //  Check if face is front- or back-facing
            //
            t = -numerator / denominator;
            if (denominator > 0) {
                //  Back-facing plane. Update tFar.
                if (t < tFar) {
                    tFar = t;
                }
            } else {
                //  Front-facing plane. Update tNear.
                if (t > tNear) {
                    tNear = t;
                }
            }

            //
            //  Check for invalid logical intersection
```

```
            //  of half-lines.
            //
            if (tNear > tFar) {
                return false;
            }
        }
    }

    return true;
}
```

射线或线段与多面体的相交

对于射线的情形，射线的原点可能位于多面体内。为了处理射线，可以修改上述的直线算法。检查t_{near}的值以确定它是否小于 0，如果是，则射线的原点位于相应平面所定义的半空间内。在这种情形中，再检查是否有$t_{far} < \infty$：如果是，那么t_{far}就是第一个有效的相交。

在线段的情形中，我们必须检查计算得到的t值，以确定它们是否在范围$0 \leqslant t \leqslant 1$之内。

11.3 线形对象与二次曲面的相交

二次曲面包括椭球面、圆柱面、圆锥面、双曲面和抛物面（参见 9.4 节）。本节将讨论计算线形对象与二次曲面相交的一般方法。然而，这种方法没有利用任何特殊的二次曲面的几何性质。针对特殊的二次曲面的算法的效率一般较高，我们在本节中也将讨论一些这类算法。

11.3.1 线形对象与一般二次曲面的相交

二次曲面的一般隐含形式方程为

$$q(x,y,z) = ax^2 + 2bxy + 2cxz + 2dx + ey^2 + 2fyz + 2gy + hz^2 + 2iz + j = 0 \tag{11.7}$$

该方程可以用矩阵形式表示为

$$[x \quad y \quad z \quad 1]\begin{bmatrix} a & b & c & d \\ b & e & f & g \\ c & f & h & i \\ d & g & i & j \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = 0 \tag{11.8}$$

设$X = [x \quad y \quad z \quad 1]$，就能更简洁地将二次曲面表示为

$$XQX^{\mathrm{T}} = 0 \tag{11.9}$$

计算直线$\mathcal{L}(t) = P + t\vec{d}$与二次曲面的相交时，可以将直线方程直接代入方程（11.9）中：

$$\begin{aligned} q(x,y,z) &= XQX^{\mathrm{T}} \\ &= [P + t\vec{d}\,]Q[P + t\vec{d}\,]^{\mathrm{T}} \\ &= (\vec{d}Q\vec{d}^{\mathrm{T}})t^2 + (\vec{d}QP^{\mathrm{T}} + PQ\vec{d}^{\mathrm{T}})t + PQP^{\mathrm{T}} \\ &= 0 \end{aligned}$$

这是一个形式如下的二次方程

$$at^2 + bt + c = 0$$

可用一般的形式来求解该方程

$$t = \frac{b \pm \sqrt{b^2 - 4ac}}{2a}$$

如果存在两个这种形式的实数（非虚数）根，那么直线将与二次曲面相交两次。如果存在两个具有相同值的实数根，那么直线与二次曲面相接触，而不贯穿二次曲面。如果根是虚数（此时$b^2 - 4ac < 0$），那么直线与曲面不相交。隐式定义的曲面$q(x, y, z) = 0$在曲面上的点s处的法线为q的梯度：

$$\vec{n}_S = \nabla q(S)$$

用方程（11.9）来表示，有

$$\vec{n} = 2\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} QS^{\mathrm{T}}$$

对于最后一步，可能需要进行一些解释：一般地，

$$\nabla f(x, y, z) = \left[\frac{\partial f(x, y, z)}{\partial x}, \frac{\partial f(x, y, z)}{\partial y}, \frac{\partial f(x, y, z)}{\partial z}\right]$$

也就是说，我们只需计算关于二次曲面的坐标系的基底向量的偏导数。再往回引用简洁的矩阵表示，可得

$$\frac{\partial X}{\partial x} = [1 \quad 0 \quad 0 \quad 0]$$

$$\frac{\partial X}{\partial y} = [0 \quad 1 \quad 0 \quad 0]$$

$$\frac{\partial X}{\partial z} = [0 \quad 0 \quad 1 \quad 0]$$

例如，

$$\begin{aligned} \vec{n}_x &= \frac{\partial f(x, y, z)}{\partial x} \\ &= \frac{\partial X}{\partial x} QX^{\mathrm{T}} + XQ\frac{\partial X^{\mathrm{T}}}{\partial x} \\ &= 2\frac{\partial X}{\partial x} QX^{\mathrm{T}} \\ &= 2[1 \quad 0 \quad 0 \quad 0]QX^{\mathrm{T}} \end{aligned}$$

伪码为

```
int LineQuadricIntersection(Matrix4x4 Q, Line3D l, float t[2])
{
    float a, b, c;
    Matrix dTrans = transpose(l.d);
```

```
        // * denotes matrix multiplication
        a = dTrans * Q * l.d;
        b = dTrans * Q * l.p + transpose(l.p) * Q * l.d;
        c = transpose(l.p) * Q * l.p;

        float discrm = b * b - 4 * a * c;
        if (discrm < 0) {
            return 0;
        }

        if (discrm == 0) {
            t[0] = b / (2 * a);
            return 1;
        } else {
            t[0] = (-b + sqrt(discrm)) / (2 * a);
            t[1] = (-b - sqrt(discrm)) / (2 * a);
        }
        return 2;
}
```

11.3.2 线形对象与球面的相交

为了方便本节的讨论，球面用一个球心 C 和半径 r 来表示，因此，球面的隐含形式方程为

$$f(X) = \|X - C\|^2 = r^2 \tag{11.10}$$

线形对象（用起点 P 和方向 $\hat{d}$ 来表示）和球面的相交可通过将线形对象的方程（方程（11.2））代入方程（11.10）来计算：

$$\|X - C\|^2 = r^2$$

$$\|P + t\hat{d} - C\|^2 = r^2$$

$$\|(P - C) + t\hat{d}\|^2 = r^2$$

$$(t\hat{d} + (P - C)) \cdot (t\hat{d} + (P - C)) - r^2 = 0$$

$$t^2(\hat{d} \cdot \hat{d}) + 2t(\hat{d} \cdot (P - C)) + (P - C) \cdot (P - C) - r^2 = 0$$

$$t^2 + 2t(\hat{d} \cdot (P - C)) + (P - C) \cdot (P - C) - r^2 = 0$$

这个二阶方程的形式为

$$at^2 + 2bt + c = 0$$

可以通过直接利用二次方程公式来求解该方程：

$$t = \frac{-(\hat{d} \cdot (P - C)) \pm \sqrt{(\hat{d} \cdot (P - C))^2 - 4((P - C) \cdot (P - C) - r^2)}}{2} \tag{11.11}$$

对于直线 / 球面之间的相交，有三种可能的条件，每一种条件都可以用方程（11.11）中的判别式 $(\hat{d} \cdot (P - C))^2 - 4(P - C) \cdot (P - C) - r^2)$ 的值来标识。

i．不相交：判别式为负，此时根为虚根。

ii．一次相交：直线与球面相切，此时判别式等于零。

iii．两次相交：判别式大于零。

图 11.9 显示了这三种可能的构形，以及线形对象是射线且起点位于球面的情形。在最后一种情形中，存在两次相交，可是，从数学的观点来说，只有一次相交位于射线的范围内。

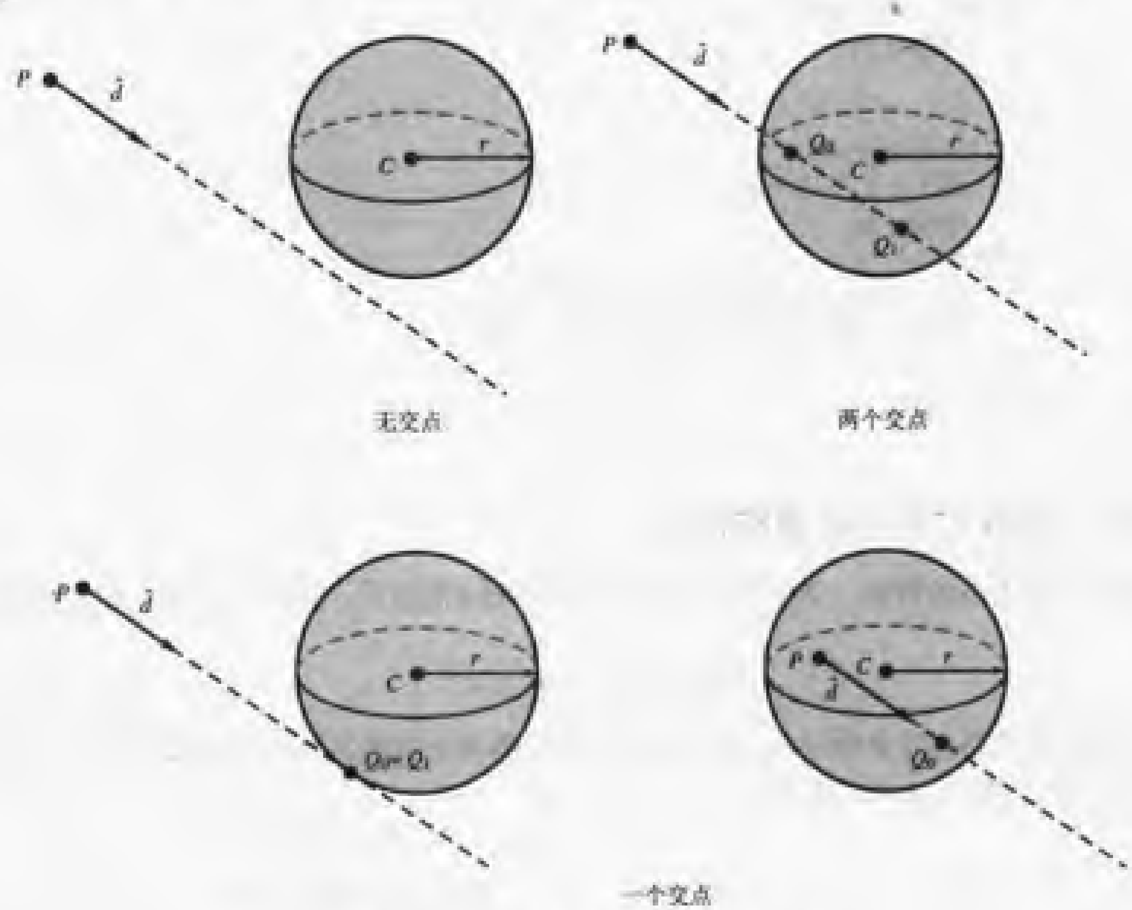


图 11.9　射线与球面之间的可能相交

伪码为

```
int LineSphereIntersection(Sphere sphere, Line3d line, float t[2])
{
    float a, b, c, discrm;
    Vector3D pMinusC = l.origin - sphere.center;
    a = Dot(line.direction, line.direction);
    b = 2 * Dot(l.direction, pMinusC);
    c = Dot(pMinusC, pMinusC) - sphere.radius * sphere.radius;
    discrm = b * b - 4 * a * c;
    if (discrm > 0) {
        t[0] = (-b + sqrt(discrm)) / (2 * a);
        t[1] = (-b - sqrt(discrm)) / (2 * a);
        return 2;
    } else if (discrm == 0) {
        // The line is tangent to the sphere
```

```
            t[0] = -b / (2 * a);
            return 1;
        } else {
            return 0;
        }
    }
```

射线或线段与球面的相交

对于射线的情形，必须检查参数值或相交的值（如果判别式为非负的）是否为非负的。检查相交是否存在的一种费时不多的方法是，判断射线的起点P是否位于球面内，如果项$(P-C)\cdot(P-C)-r^2$是非正的，那么点P位于球面内。

在线段的情形中，可以使用相同的方法，但是我们检查根的范围是否在 0 和 1 之间。Paul Bourke（1992）建议使用一种效率更高的方法：直线P_0P_1上距C最近的点位于从C到直线的一条垂直线上；换句话说，如果Q是直线上的最接近点，那么

$$(C-Q)\cdot(P_1-P_0)=0$$

将直线方程代入该式，可得

$$(C-P_0-u(P_1-P_0))\cdot(P_1-P_0)=0$$

如果我们求解该方程，可得

$$u=\frac{(C-P_0)\cdot(P_1-P_0)}{(P_1-P_0)\cdot(P_1-P_0)}$$

如果$u<0$或者$u>1$，那么，最接近的点位于线段的范围内。如果存在一次相交（或多次相交），那么$u\leqslant r$。如果这两种测试都成立，那么我们就能像前面一样计算出确切的相交。

11.3.3 线形对象与椭球面的相交

本节介绍线形对象与椭球面的相交，如图 11.10 所示。

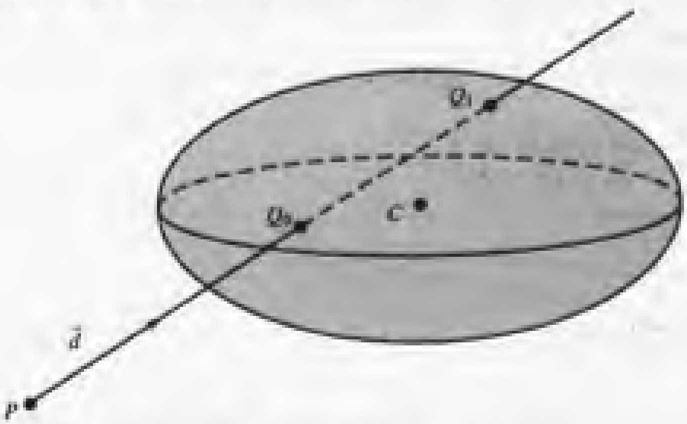


图 11.10 线形对象与椭球面的相交

线形对象$\mathcal{L}$表示为一般的形式，即用原点P和方向$\vec{d}$来表示：

$$\mathcal{L}(t)=P+t\vec{d}$$

椭球面可以用一个中心 C，一个半径 r，以及三个缩放因子来表示——每一个都与一个基底向量相关。其隐含形式方程为

$$k(x-C_x)^2+l(y-C_y)^2+m(z-C_z)^2-r^2=0 \tag{11.12}$$

如果我们将（坐标形式的）直线方程代入方程（11.12）中，可得

$$k(P_x+td_x-C_x)^2+l(P_y+td_y-C_y)^2+m(P_z+td_z-C_z)^2-r^2=0$$

这是一个形式如下的二次方程

$$at^2+bt+c=0$$

其中

$$a=kd_x^2+ld_y^2+md_z^2$$

$$b=2k(P_x-C_x)d_x+2l(P_y-C_y)d_y+2m(P_z-C_z)d_z$$

$$c=k(P_x-C_x)^2+l(P_y-C_y)^2+m(P_z-C_z)^2$$

与球面的情形一样，相交的次数取决于判别式的值（参见 11.3.2 节）。

伪码为

```
int LineEllipsoidIntersection(Ellipsoid e, line3D l, float t[2])
{
    float a, b, c, discrm;
    int numSoln = 0;
    a = e.k * l.d.x^2 + e.l * l.d.y^2 + e.m * l.d.z^2;
    b = 2 * k * (l.p.x - e.center.x) * l.d.x  + 2 * l * (l.p.y -
            e.center.y) * l.d.y + 2 * m * (l.p.z - e.center.z) * l.d.z;
    c = k * (l.p.x - e.center.x)^2 + l * (l.p.y - e.center.y)^2 +
            m * (l.p.z - e.center.z)^2;

    discrm = b * b - 4 * a * c;
    if (discrm > 0) {
        t[numSoln] = (-b + sqrt(discrm)) / (2 * a);
        numSoln++;
        t[numSoln] = (-b - sqrt(discrm)) / (2 * a);
        numSoln++;
    } else if (discrm == 0) {
        t[0] = -b / (2 * a);
        numSoln++;
    }

    return numSoln;
}
```

另一种方法以如下的思想为基础，即椭球面就是球面通过（不均衡）缩放变化来形成的。如果我们逆转该变换，那么椭球面将还原为球面。如果相同的逆变换应用于线形对象，那么我们就能找到线形对象与球面的相交，然后将交点（如果存在的话）变换回去。为了简化一点，我们可以将椭球面平移到原点。

对于缩放因子为 k，l 和 m，以及中心为 C 的椭球面，将椭球面变换成球面的矩阵M为

$$\mathbf{M}=\begin{bmatrix} \frac{1}{k} & 0 & 0 & 0 \\ 0 & \frac{1}{l} & 0 & 0 \\ 0 & 0 & \frac{1}{m} & 0 \\ -C_x & -C_y & -C_z & 1 \end{bmatrix}$$

于是，线形对象为

$$\mathcal{L}'(t)=P\mathbf{M}+t\vec{d}\mathbf{M}$$

那么椭球面为

$$x^2+y^2+z^2-r^2=0$$

这就得到一个相交方程

$$(P_x+td_x)^2+(P_y+td_y)^2+(P_z+td_z)^2-r^2=0$$

可用线形对象 / 球面的相交算法来求解该方程（注意，当球面的中心位于原点时，效率可能更高一些，这一点值得利用）。

伪码为

```
int LineEllipsoidIntersection(Ellipsoid e, line3D l, Point3D Intersection[2])
{
    float a, b, c, discrm;
    int numSoln;

    Line3D transformedl;
    Sphere s;
    Matrix4x4 M, MInv;
    M = e.TransformMatrix();
    transformedl.d = l.d * M;
    transformedl.p = l.p * M;
    s.radius = e.radius;
    float t[2];
    numSoln = lineSphereIntersection(s,transformedl,t)
    if (numSoln > 0) {
        MInv = M.Inverse();
    }
    for (i = 0 ; i < numSoln ; i++) {
        Intersection[i] = (transformedl.p + t[i] * transformedl.d ) * MInv;
    }
    return numSoln;
}
```

最后一点关于表示法的说明：经常可以看到如下形式的椭球面方程

$$\left(\frac{x-C_x}{a}\right)^2+\left(\frac{y-C_y}{b}\right)^2+\left(\frac{z-C_z}{c}\right)^2-r^2=0$$

这与我们在此使用的形式等价，即 $k=1/a^2, l=1/b^2$ 和 $m=1/c^2$ 。如果我们使用这种表示法，那么我们可以直接写出椭球面的参数形式方程：

对于缩放因子为 k, l 和 m，以及中心为 C 的椭球面，将椭球面变换成球面的矩阵M为

$$\mathbf{M} = \begin{bmatrix} \frac{1}{k} & 0 & 0 & 0 \\ 0 & \frac{1}{l} & 0 & 0 \\ 0 & 0 & \frac{1}{m} & 0 \\ -C_x & -C_y & -C_z & 1 \end{bmatrix}$$

于是，线形对象为

$$\mathcal{L}'(t) = P\mathbf{M} + t\vec{d}\mathbf{M}$$

那么椭球面为

$$x^2 + y^2 + z^2 - r^2 = 0$$

这就得到一个相交方程

$$(P_x + td_x)^2 + (P_y + td_y)^2 + (P_z + td_z)^2 - r^2 = 0$$

可用线形对象 / 球面的相交算法来求解该方程（注意，当球面的中心位于原点时，效率可能更高一些，这一点值得利用）。

伪码为

```
int LineEllipsoidIntersection(Ellipsoid e, line3D l, Point3D Intersection[2])
{
    float a, b, c, discrm;
    int numSoln;

    Line3D transformedl;
    Sphere s;
    Matrix4x4 M, MInv;
    M = e.TransformMatrix();
    transformedl.d = l.d * M;
    transformedl.p = l.p * M;
    s.radius = e.radius;
    float t[2];
    numSoln = lineSphereIntersection(s,transformedl,t)
    if (numSoln > 0) {
        MInv = M.Inverse();
    }
    for (i = 0 ; i < numSoln ; i++) {
        Intersection[i] = (transformedl.p + t[i] * transformedl.d ) * MInv;
    }
    return numSoln;
}
```

最后一点关于表示法的说明：经常可以看到如下形式的椭球面方程

$$\left(\frac{x - C_x}{a}\right)^2 + \left(\frac{y - C_y}{b}\right)^2 + \left(\frac{z - C_z}{c}\right)^2 - r^2 = 0$$

这与我们在此使用的形式等价，即 $k = 1/a^2, l = 1/b^2$ 和 $m = 1/c^2$。如果我们使用这种表示法，那么我们可以直接写出椭球面的参数形式方程：

我们将给出计算线形对象与用参数标准表示法定义的圆柱面相交的算法。这种算法既可以处理普通的一般表示法定义的圆柱面，也能用来处理标准表示法定义的圆柱面，而且也确实能对这种算法进行争论。你也可以争辩，如果你已将标准柱体变换到其世界空间位置，然后，例如，让其与射线相交，这将不需要将射线变换到圆柱面所在的本地空间，然后再将交点变换回世界空间。所有这些都可能成立，但是几乎在所有提供圆柱面（以及圆锥面等）的应用程序和工具库中，几何对象都是用场景图来组织的，其中的变换沿继承树继承下来。因为像圆柱面这样的对象是通过变换串来定位、确定大小和方向的，因此该对象也可以从标准位置变换而来。

参数标准表示法

我们在这里用其位于原点的基底、其与（本地）z 轴对齐的轴、一个指定的半径 r 和高度 h 来表示圆柱面。在这种情形中，圆柱体的方程为

$$x^2+y^2=r^2, \qquad 0\leqslant z\leqslant h \tag{11.13}$$

圆柱面的端面分别位于平面 $z=0$ 和 $z=h$ 上，它们的方程为

$$x^2+y^2\leqslant r^2, \qquad z=0, z=h$$

为了使直线 $\mathcal{L}(t)=P+t\vec{d}$ 与上述的圆柱面相交，我们需要用 $\mathbf{M}^{-1}$ 来变化 $\mathcal{L}$（将其变换回标准圆柱面所在的本地空间），并计算交点的参数 t，然后将 t 代回直线方程以计算以直线的坐标系为参照的真实交点。

计算直线与面的交点非常简单：将直线方程代入方程（11.13），可得

$$(P_x+td_x)^2+(P_y+td_y)^2-r^2=0$$

扩展并合并项，可得

$$(d_x^2+d_y^2)t^2+2(d_xP_x+d_yP_y)t+(P_x^2+P_y^2)-r^2=0$$

与前面的情形一样，这也是具有如下形式的二次方程

$$at^2+bt+c=0$$

可用如下的二次公式来求解

$$t=\frac{b\pm\sqrt{b^2-4ac}}{2a}$$

一旦计算出 t，就能将其代回直线方程以计算交点（也是位于以圆柱面为参照的本地坐标系中）。注意，如果交点存在，那么交点就是直线与无限圆柱面的交点。为了计算直线与被端面所包围的圆柱面部分的交点，还需要用条件 $0\leqslant z\leqslant h$ 来测试得到的交点。

通过计算直线与端面所在的平面的交点，然后检测它是否满足 $x^2+y^2\leqslant r^2$，就能得到直线与端面的交点。最接近的交点就是所有有效的交点中最接近的一个。

伪码为

```
bool LineCylinderIntersection(Cylinder c, Line3D l, Point3D closestIntersection)
{
    Matrix4x4 transform, invTransform;
```

```
float a, b, c, discrm;
float t[4];
bool valid[4];
Line3D tline;
Point3D ipoint[4];
transform = c.transformMatrix();
invTransform = transform.Inverse();
tline.direction = l.direction * invTransform;
tline.origin = l.origin * invTransform;

a = (tline.direction.x + tline.direction.y) ^ 2;
b = 2 * (tline.direction.x * tline.origin.x + tline.direction.y * tline.origin.y);
c = tline.origin.x^2 + tline.origin.y^2 - c.radius^2;

discrm = b*b - 4*a*c;

if (discrm > 0) {
    t[0] = (-b + sqrt(discrm) / (2 * a);
    t[1] = (-b - sqrt(discrm) / (2 * a);
    ipoint[0] = tline.origin + t[0] * tline.direction;
    if (ipoint[0].z < 0 || ipoint[0].z > c.height) {
        valid[0] = false;
    } else {
        valid[0] = true;
    }
    ipoint[1] = tline.origin + t[1] * tline.direction;
    if (ipoint[1].z < 0 || ipoint[1].z > c.height) {
        valid[1] = false;
    } else {
        valid[1] = true;
    }

    // Check end caps
    Plane3D p1,p2;
    p1.normal = [0,0,1];
    p2.normal = [0,0,-1];
    p1.point = [0,0,0];
    p2.point = [0,0,c.height];
    if (lineIntersectPlane(p1,tline,t[3])) {
        ipoint[3] = tline.origin + t[3] * tline.direction;
        float d = ipoint[3].x^2 + ipoint[3].y^2;
        if (d < = c.radius^2 ) {
            valid[3] = true;
        } else {
            valid[3] = false;
        }
    } else {
        valid[3] = false;
    }
```

```
        if (lineIntersectPlane(p2,tline,t[4])) {
            ipoint[4] = tline.origin + t[4] * tline.direction;
            float d = ipoint[4].x^2 + ipoint[4].y^2;
            if (d < = c.radius^2) {
                valid[4] = true;
             } else {
                valid[4] = false;
             }
        } else {
            valid[4] = false;
        }

        // Find smallest t of the valid intersection points
        float mint = infinity;
        int minIndex = 0;
        bool hit = false;
        for (i = 0; i < 4 ; i++) {
            if (valid[i]) {
                if (t[i] < mint) {
                    mint = t[i];
                    minIndex = i;
                    hit = true;
                }
            }
        }

        closestIntersection = transform*ipoint[minIndex];
        return hit;
    } else if (discrm == 0) {
        // Ray is tangent to side, no need to check caps
        t[0] = -b / (2 * a);
        ipoint[0] = tline.origin + t[0] * tline.direction;
        if (ipoint[0].z > c.z || ipoint[0].z < 0) {
            return 0;
        }

        closestIntersection = transform*ipoint[0];
        return true;
    }

    return false;
}
```

11.3.5　线形对象与圆锥面的相交

与圆柱面一样，圆锥面也能用不同的标准形式或一般形式来表示（如图 11.13 和图 11.14 所示）。

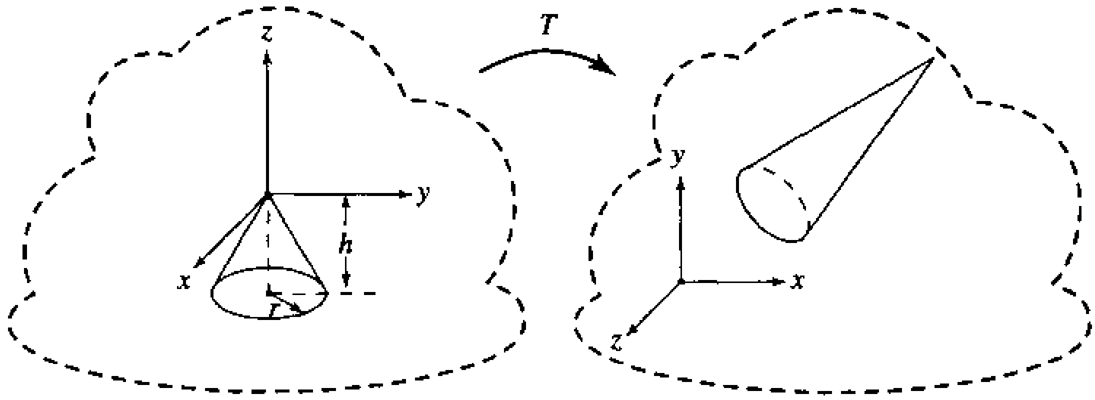


图 11.13　圆锥面的参数标准表示法

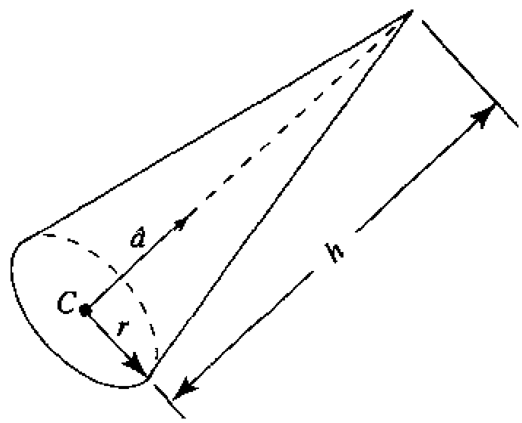


图 11.14　圆锥面的一般表示法

参数标准表示法

标准圆锥面的方程为

$$\frac{x^2}{k^2}+\frac{y^2}{k^2}=z^2$$

其中

$$-h\leqslant z\leqslant 0$$

$$k=\frac{r}{h}$$

其底面可表示为

$$z=-h$$

$$x^2+y^2\leqslant r^2$$

设直线的方程为$X(t)=P+t\vec{d}$，其中$t\in\mathbb{R}$。圆锥面具有顶点V，轴方向向量$\hat{a}$，以及轴与外边之间的夹角θ。在许多应用中，圆锥面是锐角的，即$\theta\in(0,\pi/2)$。实际上，我们在本节中假设圆锥面是锐角的，因此$\cos\theta>0$。圆锥面包含满足如下条件的点X，即$X-\vec{v}$与$\hat{a}$之间的夹角为θ。用代数表达式来表示为

$$\hat{a}\cdot\left(\frac{X-V}{\|X-V\|}\right)=\cos\theta$$

图 11.15 显示了上述圆锥面的二维表示。阴影部分表示圆锥面的里面，将上式中的"="换成"≥"就得到这部分区域的代数表达式。

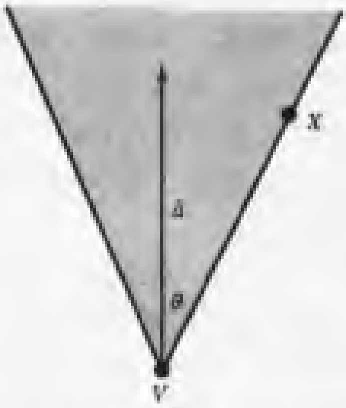


图 11.15 一个锐角锥，内部区域用阴影表示

为了避免$\|X-V\|$的平方根运算，可以对方程的两边平方，以得到如下的二次方程

$$(\hat{a}\cdot(X-V))^2=(\cos^2\theta)\|X-V\|^2$$

然而，满足该方程的点集构成一个双锥面。原来的圆锥面在平面$\hat{a}\cdot(X-V)=0$的$\hat{a}$所指向的那一边。上述二次方程定义了原来的圆锥面和其沿上述平面的反射圆锥面。特别地，如果X是二次方程的一个解，那么它沿顶点的反射，即$2V-X$，也是方程的一个解。图 11.16 显示了一个双锥面。为了去掉反射圆锥面，二次方程的解还应该满足$\hat{a}\cdot(X-V)\geq 0$。同时，这个二次方程也可以改写为如下的二次方程形式，即$(X-V)^{\mathrm{T}}\mathbf{M}(X-V)=0$，其中$\mathbf{M}=(\hat{a}\hat{a}^{\mathrm{T}}-\gamma^2 I)$且$\gamma=\cos\theta$。因此，$X$是锐角圆锥面上的点，并满足

$$(X-V)^{\mathrm{T}}\mathbf{M}(X-V)=0 \text{ 和 } \hat{a}\cdot(X-V)\geq 0$$

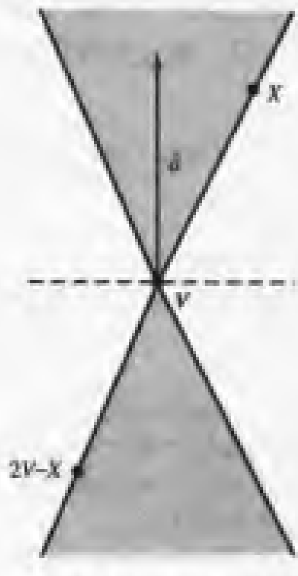


图 11.16 一个锐角双锥，内部区域用阴影表示

为了找到直线与圆锥面之间的交点，将$X(t)$代入二次方程，化简得到$c_2t^2+2c_1t+c_0=0$，其中$\vec{\Delta}=P-V$，$c_2=\vec{d}^{\mathrm{T}}\mathrm{M}\vec{d}$，$c_1=\vec{d}^{\mathrm{T}}\mathrm{M}\vec{\Delta}$且$c_0=\vec{\Delta}^{\mathrm{T}}\mathrm{M}\vec{\Delta}$。下面的讨论分析了上述二次方程，以确定直线与双锥面的交点。必须对求得的交点应用点积测试，以排除在反射圆锥面上的交点。

上述的二次方程可能因$c_2=0$而退化。在这种情形中，方程变成线性方程，而且，即使在这种情形下，方程还可能因$c_1=0$而进一步退化。计算交点的算法必须考虑这类情形。

首先，假设$c_2\neq 0$。定义$\delta=c_1^2-c_0c_2$。如果$\delta<0$，那么二次方程没有实数根，此时直线与双锥面不相交。如果$\delta=0$，那么二次方程具有两个相等的实数根$t=-c_1/c_2$，此时直线与双锥面相切于一个点。如果$\delta>0$，那么二次方程具有两个不相等的实数根$t=(-c_1\pm\sqrt{\delta})/c_2$，此时直线与双锥面相交于两个点。

其次，假设$c_2=0$。这意味着$\vec{d}^{\mathrm{T}}\mathrm{M}\vec{d}=0$。结果是，对所有的$s\in\mathbb{R}$，直线$V+s\vec{d}$都位于双锥面上。从几何意义上来说，直线$P+t\vec{d}$与圆锥面上的一些直线平行。如果再有$c_0=0$，则$\vec{\Delta}^{\mathrm{T}}\mathrm{M}\vec{\Delta}=0$。结果是，$P$是双锥面上的一个点。即使如此，直线也不一定完全包含于圆锥面内（虽然也有可能）。图 11.17 显示了$c_2=0$及$c_0\neq 0$或$c_0=0$的情形。

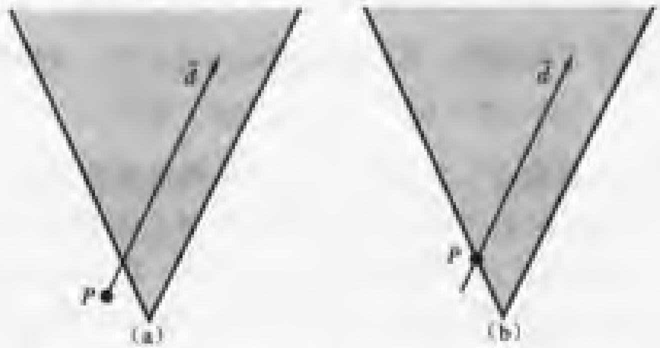


图 11.17　$c_2=0$ 的情形，(a) $c_0\neq 0$；(b) $c_0=0$

最后，如果对所有的i，都有$c_i=0$，那么，对所有的$t\in\mathbb{R}$，$P+t\vec{d}$都位于双锥面上。用代数式来表示就是：如果$c_1\neq 0$，则直线方程的根为$t=-c_2/(2c_1)$。如果$c_1=0$且$c_2\neq 0$，则直线不与圆锥面相交。如果$c_1=c_2=0$，那么直线包含于双锥面内。

伪码为

```
bool LineConeIntersection(Line3D line, Cone3D cone, Point3D closestIntesection)
{
    Vector3D axis = cone.axis;
    float cosTheta = cos(cone.theta);
    Matrix4x4 M = axis * axis.transpose() - cosTheta * cosTheta * Matrix4x4::Identity;
    float c2, c1, c0, discrm;

    Vec3D delta = line.origin - cone.vertex;
    c2 = line.direction.transpose() * M * line.direction;
    c1 = line.direction.transpose() * M * delta;
    c0 = delta.transpose() * M * delta;

    discrm = c1 * c1 - c2 * c0;
```

```
if (discrm > 0) {
    float    t[3];
    Point3D ipoint[3];
    int      minIndex;
    bool     valid[3];

    if (fabs(c2) < zeroEpislon) {
        if (fabs(c1) < zeroEpislon) {
            valid[0] = false;
            valid[1] = false;
        } else {
            t[0] = -c0 / (2 * c1);
            ipoint[0] = line.origin + t[0] * line.direction;
        }
    } else {
        t[0] = (-c2 + sqrt(discrm)) / c0;
        t[1] = (-c2 - sqrt(discrm)) / c0;

        ipoint[0] = line.origin + t[0] * line.direction;
        ipoint[1] = line.origin + t[1] * line.direction;

        float  scalarProjection;

        if (scalarProjection = Dot(axis,ipoint[0] - cone.vertex) < 0) {
            valid[0] = false;
        } else {
            if (scalarProjection > cone.height) {
                valid[0] = false;
            } else {
                valid[0] = true;
            }
        }

        if (scalarProjection = Dot(axis,ipoint[1] - cone.vertex) < 0) {
            valid[1] = false;
        } else {
        if (scalarProjection > cone.height) {
            valid[1] = false;
        } else {
            valid[1] = true;
        }
    }
}

// Check for earlier intersection with cap
Plane3D pl;
pl.normal = axis;
pl.origin = cone.vertex + cone.height * axis;

if (lineIntersectPlane(pl, line, t[3])) {
```

```
            ipoint[3] = line.origin + t[3] * line.direction;
            if (distance(pl.origin, ipoint[3]) <= cone.radius) {
                valid[3] = true;
            } else {
                valid[3] = false;
            }
        } else {
            valid[3] = false;
        }

        // Now find earliest valid intersection
        bool hit = false;
        int minIndex = 0;
        float mint = infinity;

        for (i = 0 ; i < 3 ;i++) {
            if (valid[i]) {
                if (t[i] < mint) {
                    mint = t[i];
                    minIndex = i;
                    hit = true;
                }
            }
        }
        closestIntersection = ipoint[minIndex];
        return hit;
    } else if (discrm == 0) {
        // No need to check cap
        float scalarProjection;
            t[0] = -c1 / c2;
            ipoint[0] = line.origin + t[0] * line.direction;

            if (scalarProjection = dotProd(axis, ipoint[0] - cone.vertex) >= 0) {
                if (scalarProjection <= cone.height) {
                    closestIntersection = ipoint[0];
                    return true;
                }
            }
        }
        return false;
    }
```

11.4 线形对象与多项式曲面的相交

多项式曲面可以表示为向量值函数$X : D \subset \mathbb{R}^2 \rightarrow \mathbb{R}^3$，比如$X(s,t)$，该函数的定义域为$D$，值域为$R$。$X(s,t)$的每一个分量$X_i(s,t)$都是具有特定参数的一个多项式

$$X_i(s,t) = \sum_{j=0}^{n_i} \sum_{k=0}^{m_i} a_{ijk} s^j t^k \tag{11.14}$$

其中$n_i + m_i$为多项式的次数。定义域D一般为$\mathbb{R}^2$或$[0,1]^2$。有理多项式曲面可表示为一个向量值函数$X(s,t)$，每一个分量$X_i(s,t)$都是一个有理多项式

$$X_i(s,t) = \frac{\sum_{j=0}^{n_i}\sum_{k=0}^{m_i} a_{ijk}s^j t^k}{\sum_{j=0}^{p_i}\sum_{k=0}^{q_i} b_{ijk}s^j t^k}$$

其中$n_i + m_i$为分子多项式的次数。$p_i + q_i$为分母多项式的次数。

在计算机图形学中常见的一些多项式曲面包括贝塞尔曲面、B 样条曲面和非均匀有理 B 样条（NURBS）曲面。

为了方便下面的讨论，用常用的形式将线形对象表示为原点和一个方向

$$\mathcal{L}(t) = P + t\vec{d} \tag{11.15}$$

其中，对于直线，$-\infty \leqslant t \leqslant \infty$；对于射线，$0 \leqslant t \leqslant \infty$；对于线段$\{P_0, P_1\}$，有$\vec{d} = P_1 - P_0$及$0 \leqslant t \leqslant 1$。

需要计算多项式曲面与线形对象相交的两种最常见的情形是，在渲染——特别是射线跟踪中，以及在处理选择或选取（用户用鼠标或其他的定位设备来选定对象）的交互应用程序中。为了说明这类应用，我们集中讨论射线－曲面的相交。图 11.18 显示了一个例子。

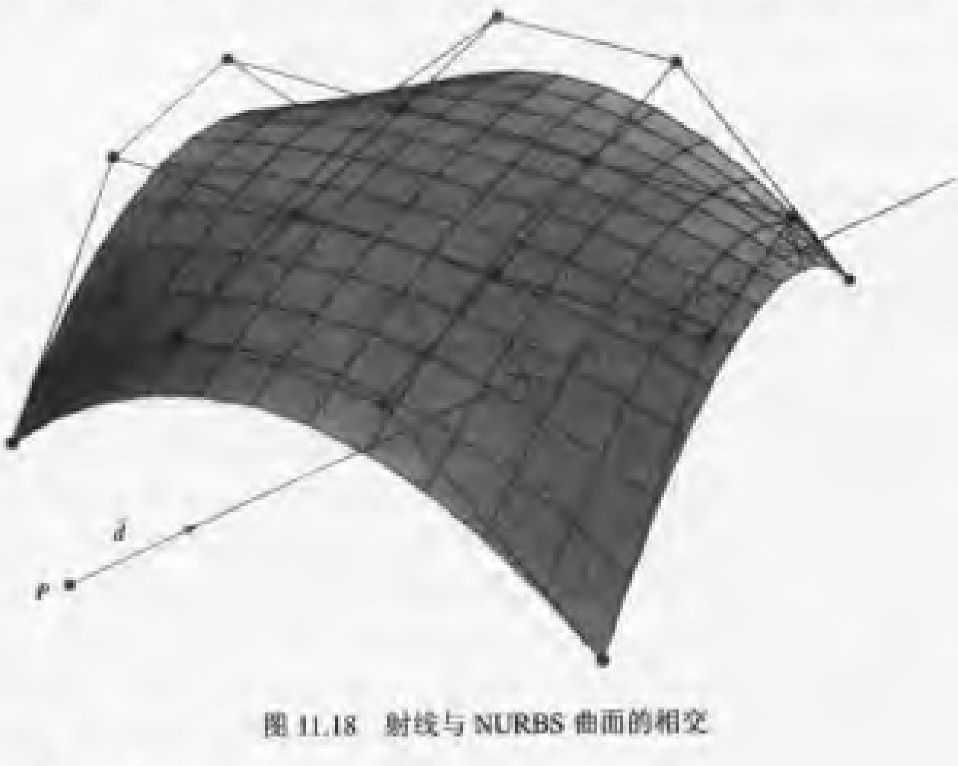


图 11.18　射线与 NURBS 曲面的相交

11.4.1　代数曲面

一般地，代数曲面用如下形式的方程来表示

$$f(x,y,z) = 0 \tag{11.16}$$

其中，函数f是一个多项式，即

$$f(x,y,z)=\sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n}a_{ijk}x^iy^jz^k$$

其次数为各分量的次数之和：$d=l+m+n$。

如果我们将方程（11.15）改写为其分量的形式，有

$$x=P_x+td_x$$

$$y=P_y+td_y$$

$$z=P_z+td_z$$

很容易看出，如果我们将上述方程代入方程（11.16），就得到另一个形式的多项式方程，如下所示

$$g(t)=\sum_{i=0}^{d}a_it^i$$

可以用标准方法来求解这个方程。注意，可能的方程根数量的最大值与多项式曲面的次数相同。实际上，有一些简单的求解方法可用来求解次数小于等于 4 的方程。对于次数更高的方程，需要使用数值方法。Hanrahan（1983）使用一种方法来求第一个孤立根（Collins 和 Akritas 1976；Collins 和 Loos 1982），然后应用 *regula falsi*。

11.4.2 自由形态曲面

自由形态的参数曲面一般定义为方程（11.14）所示的形式。线形对象与多项式笛卡尔积小曲面的相交是一个次数为 $2\times M^2$的多项式方程，其中 M 为曲面的次数。对于双三次小曲面，我们将得到一个次数为 18 的多项式相交方程。利用一种直接的求根方法（比如牛顿迭代法）可以得到一个非常慢的算法，这种算法在有的情形中还可能收敛失败。在任何情形中，只有当初始估值与第一个根相差不大时，才能得到可以预测的结果。

Kajiya（1982）提出了一种方法，后来的研究者大都采用这种方法，值得在此简单介绍一下：一条射线可以看成是两个（不共面的）平面之间的相交。为了求得射线与小曲面的相交，将曲面方程代入平面方程，据此可得到两个定义代数曲线的方程，该代数曲线由小曲面与上述的两个平面相交而得到。这两条曲线的交点给出了射线与小曲面的交点的 (u,v) 参数。然后，为了求解方程，Kajiya 利用 Laguerre 方法来求根。他发现，这种方法在稳定性和收敛性方面都优于牛顿迭代法，并且在三次方时收敛。其他的一些人也利用了这种将射线看成是两个平面的相交的方法（Sweeney 和 Bartels 1986; Martin 等 2000）。Kajiya（1982）给出了这种方法的有效性的一种完整的证明。

另一种方法是简单地将曲面细分或镶嵌成多边形（一般是三角形或四边形），然后分别求射线与每一个小碎片的相交。这种方法很容易编程实现，但是具有如下的几个问题：

- 如果没有使用某种空间划分方法（比如空间分级包围法），这种方法可能是非常低效的。
- 如果细分不够精细，那么很可能将漏掉一个相交（如图 11.19 所示）。
- 如果细分太过精细，那么将进行冗余的运算，然而，精度随着细分的精细而增加，

因此，存在一种与目标的内在不兼容性。

一般来说，直接的计算方法可能非常慢，但是非常精确，而简单的细分方法可能效率高，但是在实现这种效率时可能要牺牲精度。你可能很自然地想到，是不是可以创建一种混合这两种方法的新方法，而事实上，这种方法也确实取得了好的结果。

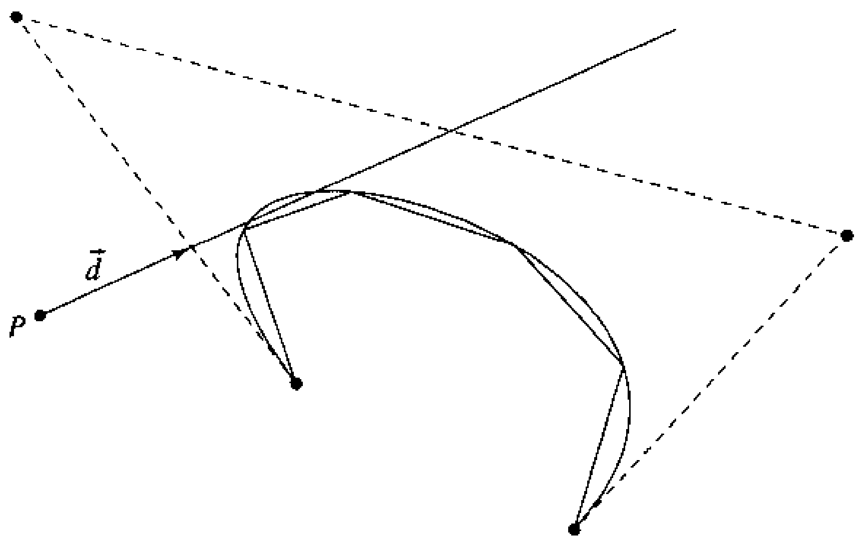


图 11.19 由于曲面嵌片数量的不足导致相交计算的失败（为清楚起见，仅显示其横截面）

在描述这类混合方法之前，还有一些其他的相交方法值得一提。Nishita，Sederberg 和 Kakimoto（1990）描述了一种它们称之为“贝塞尔裁剪”的方法。射线被看成两个正交平面的相交，而贝塞尔曲面投影到一个与射线垂直的平面上。该射线被投影到一个点，同时，这两个平面被投影到两条垂直的直线上；这样就得到一个正交基。计算（投影得到的）控制点之间的距离和“基底向量”，并且用 de Casteljau 算法来裁剪小曲面——将不再考虑不可能包含相交的曲面部分。一旦连续裁剪的小曲面的大小达到指定的阈值，交点就被认为是这个足够小的小曲面的中点。

Fournier 和 Buchanan（1984）描述了一种利用切比雪大多项式来表示多项式曲面和建立细小的有界箱的方法。将小曲面适应性地细分成大量的双线性小片，并以此来近似小曲面。这些双线性小片被组织在一个有界箱层级中，以加快计算与射线相交的速度。有界箱层级被遍历到与射线相交的叶子节点上，而且射线与双线性小片在该叶子节点上的相交就可以当做射线与曲面的（近似）相交。

Campagna，Slusallek 和 Seidel（1997）分析了上述两种方法。他们的结果显示，你可能也已经猜测到，贝塞尔裁剪法比切比雪夫分箱法要慢一些（25%~30%）。他们还注意到，切比雪夫分箱法只能处理整数数目的小片。基于这些分析结论，他们建立了自己的空间分箱层级法，他们的方法能处理有理数数目的小片，并且具有与切比雪夫分箱法相当的运算速度。

Toth（1985）描述了一种也是基于 Kajiya 方法的射线相交算法。这种方法也使用牛顿迭代法，并利用区间分析技术来解决提供有效的初始估值的问题。

我们这里提供的算法的一般结构已被 Martin 等人（2000）、Sweeney 和 Bartels（1986），以及 Campagna，Slusallek 和 Seidel（1997）使用过，我们将提到这一点。

1. 计算射线与参数多项式曲面之间的相交

可以在你选择的基中表示多项式曲面。在这里，我们使用贝塞尔基，因为任何多项式都能转换到这种基中，而且我们可以在我们的算法中利用贝塞尔基的一些性质。

我们的射线被定义为常见的形式：

$$\mathcal{L}(t) = P + t\vec{d}$$

根据 Kajiya（1982），我们将该射线表示为两个平面之间的相交：

$$\mathcal{P}_0 : a_0x + b_0y + c_0z + d_0 = 0$$

$$\mathcal{P}_1 : a_1x + b_1y + c_1z + d_1 = 0$$

如果我们设

$$\vec{n}_0 = [a_0 \quad b_0 \quad c_0]$$

$$\vec{n}_1 = [a_1 \quad b_1 \quad c_1]$$

那么（根据 Martin 等 2000），我们可以将平面定义为

$$\mathcal{P}_0 : \{P | \mathbf{P}_0 \cdot [P \quad 1] = 0\}$$

$$\mathcal{P}_1 : \{P | \mathbf{P}_1 \cdot [P \quad 1] = 0\}$$

其中

$$\mathbf{P}_0 = [\vec{n}_0 \quad d_0]$$

$$\mathbf{P}_1 = [\vec{n}_1 \quad d_1]$$

如图 11.20 所示。

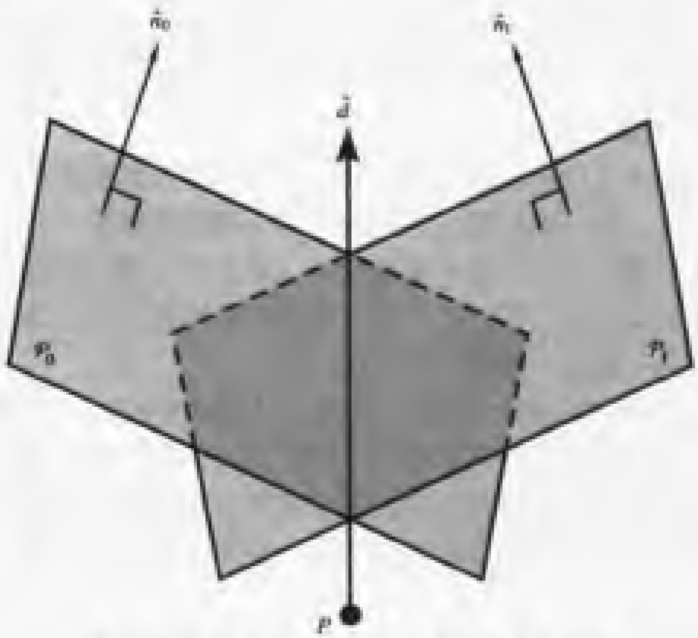


图 11.20　表示为两个平面的相交的一条射线

存在无数的包含$\mathcal{L}(t)$的平面，所有的这些平面都满足$\vec{n}_0 \perp \vec{d}$。选择$\hat{n}_0$的一种方便的方法是将n_x，n_y或n_z之一设为零，并与其余两个进行“perp”（参见 4.4.4 节）运算。一般

最好的方案就是将最大的分量设为零的方案：

$$\vec{n}_0 = \begin{cases} [d_y \quad -d_x \quad 0] & \text{如果} |d_x| > |d_y| \text{ 且 } |d_x| > |d_z| \\ [0 \quad d_z \quad -d_y] & \text{否则} \end{cases}$$

由于已知$\mathcal{P}_0$和$\mathcal{P}_1$是正交的，因此我们设

$$\vec{n}_1 = \vec{n}_0 \times \hat{d}$$

注意到$\mathcal{P}_0$和$\mathcal{P}_1$必须包含射线原点P，我们可以据此得到完整的平面方程，因而我们设

$$d_0 = -\vec{n}_0 \cdot P$$

$$d_1 = -\vec{n}_1 \cdot P$$

我们得到的（有理数）贝塞尔曲面可定义为

$$Q(u,v) = \frac{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)w_{ij}P_{ij}}{\sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)w_{ij}}$$

其中P_{ij}为贝塞尔控制点，w_{ij}为权。

通过代入，可得到平面$\mathcal{P}_k$与贝塞尔小曲面之间的相交的如下表达式：

$$\begin{aligned} S_k(u,v) &= \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(u)B_j^m(v)w_{ij}(P_{ij}\cdot[\vec{n}_k \quad d_k]) \\ &= [\vec{n}_k \quad d_k]\cdot[Q(u,v) \quad 1] \\ &= 0 \end{aligned}$$

其中$k \in \{0,1\}$。

由于射线与小曲面之间的交点都在射线上，而射线又同时在平面$\mathcal{P}_0$和$\mathcal{P}_1$上，因此，交点必定同时位于这两个平面上。所以，交点必定同时满足

$$[\vec{n}_0 \quad d_0]\cdot[Q(u,v) \quad 1] = 0$$

$$[\vec{n}_1 \quad d_1]\cdot[Q(u,v) \quad 1] = 0$$

Kajiya（1982）建议用 Laguerre 方法来求解这对隐含形式的方程，而 Martin 等人（2000）、Sweeney 和 Bartels（1986）使用牛顿迭代法。

假设我们有初始估值(u_0, v_0)，牛顿方法从这些值开始并迭代以求精确解

$$u_0 \to u_1 \to \cdots \to u_\lambda \to u_{\lambda+1} \to \cdots$$

$$v_0 \to v_1 \to \cdots \to v_\lambda \to v_{\lambda+1} \to \cdots$$

这是通过重复地求解如下的 2×2 系统

$$\begin{bmatrix} \frac{\partial S_0}{\partial u} & \frac{\partial S_0}{\partial v} \\ \frac{\partial S_1}{\partial u} & \frac{\partial S_1}{\partial v} \end{bmatrix} \begin{bmatrix} \delta u_{\lambda+1} \\ \delta v_{\lambda+1} \end{bmatrix} = \begin{bmatrix} S_0(u_\lambda, v_\lambda) \\ S_1(u_\lambda, v_\lambda) \end{bmatrix}$$

以得到如下结果来实现的

$$u_{\lambda+1} = u_\lambda - \delta u_{\lambda+1}$$

$$v_{\lambda+1} = v_\lambda - \delta v_{\lambda+1}$$

一旦满足一个或多个条件，牛顿迭代法就能得出结论，得到的解包含数对$(u_{\lambda+1}, v_{\lambda+1})$或者我们可以得出射线与小曲面不相交的结论。Martin 等人（2000），以及 Sweeney 和 Bartels（1986）都使用"成功"标准——将$(u_{\lambda+1}, v_{\lambda+1})$插入$S_0$和$S_1$，并将和与一些预定公差比较：

$$\|S_0(u_{\lambda+1}, v_{\lambda+1})\| + \|S_1(u_{\lambda+1}, v_{\lambda+1})\| < \epsilon$$

如果遇到如下的任何一个或多个条件，我们就能得出射线与曲面不相交的结论。

i．迭代超出了曲面的范围，即

$$u_{\lambda+1} < u_{\min} \text{ 或 } u_{\lambda+1} > u_{\max} \text{ 或}$$
$$v_{\lambda+1} < v_{\min} \text{ 或 } v_{\lambda+1} > v_{\max}$$

ii．迭代退化而不是增强解：

$$\|S_0(u_{\lambda+1}, v_{\lambda+1})\| + \|S_1(u_{\lambda+1}, v_{\lambda+1})\| > \|S_0(u_\lambda, v_\lambda)\| + \|S_1(u_\lambda, v_\lambda)\|$$

iii．迭代的数目达到了一些预设的限制。

2．利用封闭空间

如下的两个事实促使我们利用封闭空间方法：第一，牛顿迭代法能通过二次曲线收敛于一个根；第二，一般地，封闭空间方法已被证明对相交测试是非常有用的，因为这种方法允许快速排除，因而能够避免许多费时的相交计算。如果我们能利用封闭空间来实现双重功能，以给我们提供牛顿迭代法的初始估值，那么这些似乎无关的实现实际上就能非常有效地一起工作。

Fournier 和 Buchanan（1984）描述的切比雪夫分箱法是这种方法的一个例子，然而，正如 Campagna，Slusallek 和 Seidel（1997）所指出的，切比雪夫多项式只能用于整数数目的小曲面。Martin 等人（2000）及 Sweeney 和 Bartels（1986）都使用轴对齐有界箱，因为它们易于计算并且对计算与射线的相交非常有效。两种方法都从使用 Oslo 算法（Cohen，Lyche 和 Riesenfeld 1980；Goldman 和 Lyche 1983）使曲面变得精细的一个预处理步骤开始。在 Martin 等人（2000）的方法中，采用了一种启发式方法来估计插入的额外节点的数量。这种启发式方法考虑曲率和弧长。然后插入这些节点，每一个节点成倍增长直到在每一个方向上都达到小曲面的次数。这样就得到一个位于每一对不同的节点之间的（有理数）贝塞尔小曲面。每一个贝塞尔子曲面的$n \times m$控制网格就被包围在一个轴对齐有界箱中。这些有界箱被组织在一个层次结构中。

在 Sweeney 和 Bartels（1986）的情形中，也使用 Oslo 算法来精细曲面。他们的精细标准以精细所得的二次小曲面的大小，以及精细得到的节点是否成为"足够好的牛顿迭代法的初始估值"为基础。他们也建立了有界箱层级，但是他们的方法与 Martin 等人（2000）的方法不同：他们从在每一个精细的顶点周围建立一个有界箱开始，每一个都以某些全局定义的因子与其邻居重叠。这些重叠的有界箱然后再组合到一个封闭空间层级中。

在这两种情形中，层级的叶子节点都包含关于他们所表示的曲面区域的参数值。在 Martin 等人（2000）的方法中，叶子节点包含他们所表示的贝塞尔（子）曲面（在每一个方向上的）最小参数值和最大参数值。而在 Sweeney 和 Bartels（1986）的方法中，叶子节点包含与精细顶点相关的参数值。在这两种情形中，一旦已经确定射线与叶子节点的有界箱相交，那么(u, v)参数就都被当做牛顿迭代法的初始估值。

图 11.21 显示了对曲线使用封闭空间的基本思想，为了便于说明，图中仅显示曲线。精细的顶点显示为黑点。一旦计算出来，节点就成倍增加，以得到在每一对精细顶点之间的贝塞尔（子）曲线。于是，每一个（子）曲面的控制点都用来定义一个轴对齐有界箱，并且，一旦已经确定射线与叶子节点的有界箱相交，那么与控制点相关的参数值就被当成牛顿迭代法的初始估值。

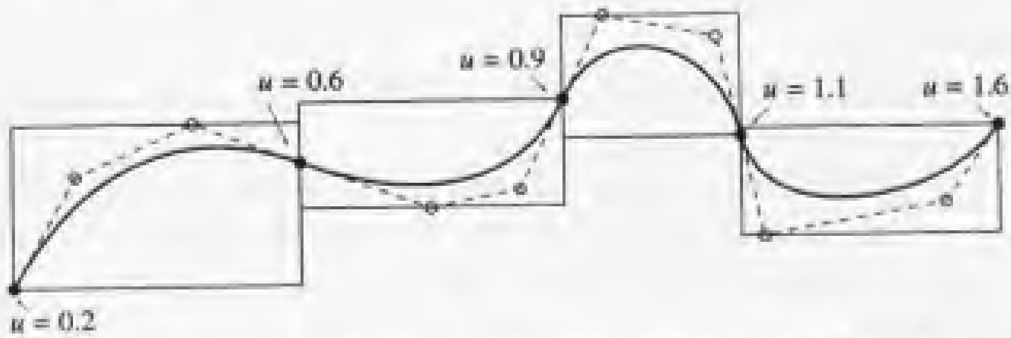


图 11.21　叶子节点有界箱由每一对精选的顶点之间的贝塞尔曲线多边形构建

图 11.22 显示了有界箱层级是如何从叶子节点建立起来的——相邻的子曲线有界箱对被组合成一个更高级的有界箱，然后这些有界箱对再被组合，依次类推，直到层级组合到一个包围整个曲面的有界箱。

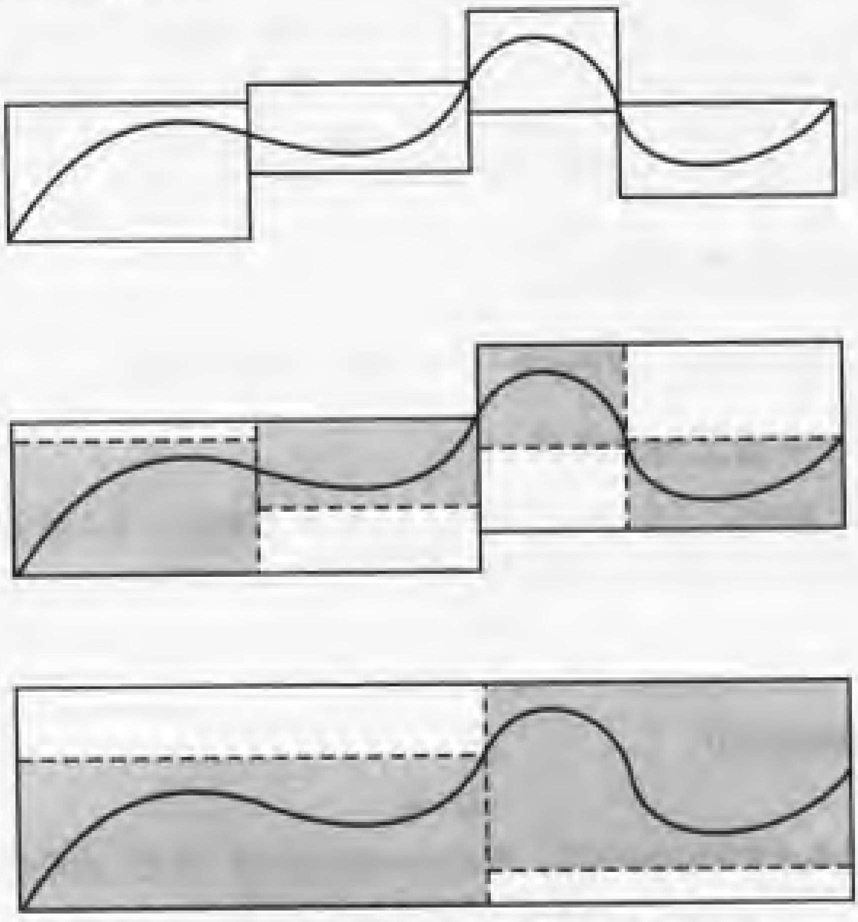


图 11.22　相邻的有界箱组合成一个下一级的有界箱

另一种使用 Oslo 算法来进行精细化的方法是采用适应性细分算法。这种方法将进行常用的折中——适应性细分模式一般将得到“更好的”曲面嵌面，在这种嵌面中，计算的点的数目和它们的相关位置将更精确地反映曲面的比例和曲率，但是却要牺牲效率（其中的预处理步骤较慢）。然而，由于有界箱层级能更好地反映曲面的几何性质，因此这种牺牲可以在实际的相交测试中得到补偿或者总体效率将更高。

这种算法的基本过程如下。

（1）预处理曲面：

a．精细或适应性地细分曲面直到满足一些平直的标准。在每一个控制点对曲面插值，存储该点的参数值。

b．对精细曲面的每一个间隔，用区域的控制点来建立一个轴对齐有界箱。将每一个箱子与该区域表示的（在每一方向上的）参数关联起来。

c．递归地将相邻的有界箱组合到一个层级中。

（2）计算相交：

a．计算射线与有界箱层级的相交。

b．当射线与一个叶子节点相交时，用与该有界箱存储在一起的参数区域来设初始估值(u_0, v_0)。一种好的选择是可以选择该区域的中点：$u_0 = (u_{\min} + u_{\max})/2$，对$v_0$做相似的处理。

c．重复应用牛顿迭代步骤直到满足收敛条件或者确定了曲线与曲面不相交为止。

Martin 等人（2000）描述了这种基本方法是如何扩展以处理整齐曲线的：分析每一个整齐曲面的方向以确定它是否定义一个洞或一个岛。然后，将每一个整齐曲线放在一个节点中，由于整齐曲线可套在一起，因此这样就建立了一个节点层级。一旦检测到射线的相交，就用交点来检测修整的层级以确定它是否被精细或返回一个交点。

11.5　平面对象之间的相交

在本节中，我们讨论计算平面对象——三角形和平面之间的相交。

11.5.1　两个平面之间的相交

两个平面如果相交（平行的平面不相交，除此之外都相交），那么相交得到的必定是一条直线$\mathcal{L}$（如图 11.23 所示）。如果我们有两个平面

$$\mathcal{P}_1 : \vec{n}_1 \cdot P = s_1$$

$$\mathcal{P}_2 : \vec{n}_2 \cdot P = s_2$$

那么相交的直线的方向为

$$\vec{n}_1 \times \vec{n}_2$$

为了完整地确定其中的相交直线，我们还需要直线上的一个点。假设这个点是$\vec{n}_1$和$\vec{n}_2$的一个线性组合，那么用这些法线向量可以表示如下的点集：

$$P = a\vec{n}_1 + b\vec{n}_2$$

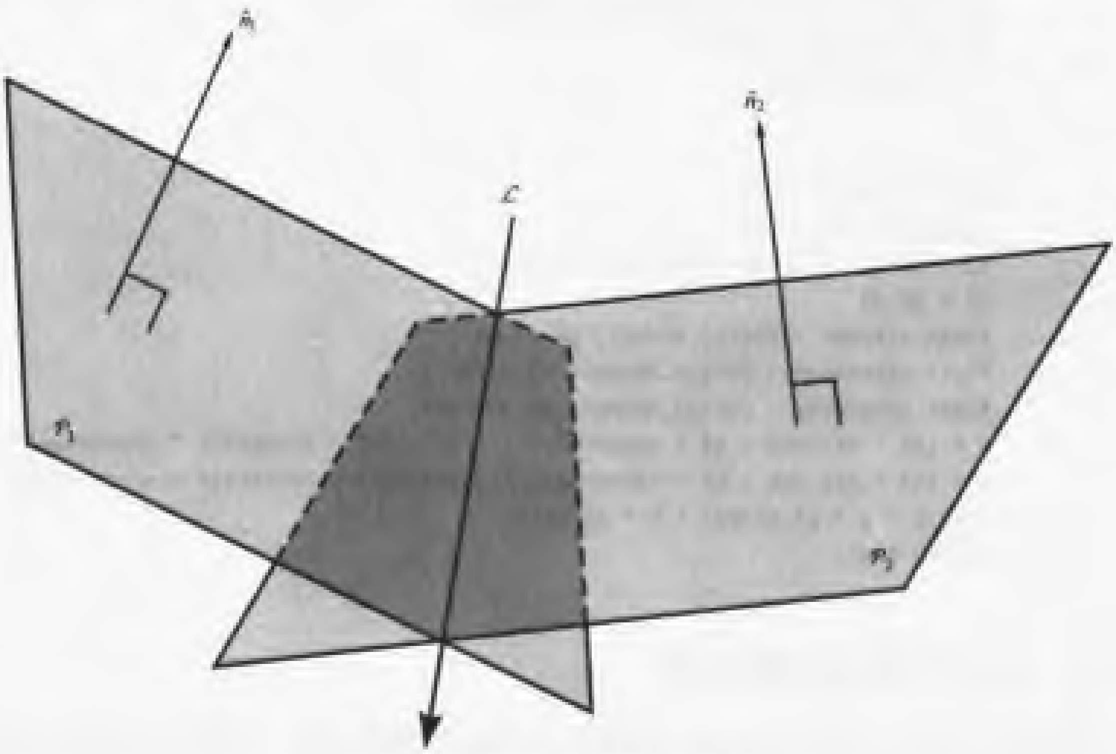


图 11.23 两个平面之间的相交

这个点必定同时位于两个平面上，因此必定同时满足这两个平面方程：

$$\vec{n}_1 \cdot P = s_1$$

$$\vec{n}_2 \cdot P = s_2$$

于是得到

$$a\|\vec{n}_1\|^2 + b\vec{n}_1 \cdot \vec{n}_2 = s_1$$

$$a\vec{n}_1 \cdot \vec{n}_2 + b\|\vec{n}_2\|^2 = s_2$$

求解 a 和 b，可得

$$a = \frac{s_2\vec{n}_1 \cdot \vec{n}_2 - s_1\|\vec{n}_2\|^2}{(\vec{n}_1 \cdot \vec{n}_2)^2 - \|\vec{n}_1\|^2\|\vec{n}_2\|^2}$$

$$b = \frac{s_1\vec{n}_1 \cdot \vec{n}_2 - s_2\|\vec{n}_1\|^2}{(\vec{n}_1 \cdot \vec{n}_2)^2 - \|\vec{n}_1\|^2\|\vec{n}_2\|^2}$$

这就得到如下的直线方程

$$\begin{aligned}\mathcal{L} &= P + t(\vec{n}_1 \times \vec{n}_2) \\ &= (a\vec{n}_1 + b\vec{n}_2) + t(\vec{n}_1 \times \vec{n}_2)\end{aligned}$$

注意，选取 $\vec{n}_2 \times \vec{n}_1$ 为直线方向也将得到相同的直线，只是方向相反。

伪码为

```
bool IntersectionOf2Planes(Plane3D p1, Plane3D p2, Line3D line)
{
```

```
        Vector3D d = Cross(p1.normal,p2.normal)
        if (d.length() == 0) {
            return false;
        }

        line.direction = d;
        float s1, s2, a, b;
        s1 = p1.d; // d from the plane equation
        s2 = p2.d;
        float n1n2dot = Dot(p1.normal, p2.normal);
        float n1normsqr = Dot(p1.normal, p1.normal);
        float n2normsqr = Dot(p2.normal, p2.normal);
        a = (s2 * n1n2dot - s1 * n2normsqr) / (n1n2dot^2 - n1normsqr * n2normsqr);
        b = (s1 * n1n2dot - s2 * n2normsqr) / (n1n2dot^2 - n1normsqr * n2normsqr);
        line.p = a * p1.normal + b * p2.normal;
        return true;
    }
```

11.5.2 三个平面之间的相交

计算三个平面之间的相交问题与计算两个平面的相交类似，但是需要考虑更多的情形，而且区分这些情形是很有用的。给定如下的三个平面

$$\mathcal{P}_0 : \{P_0, \vec{n}_0\}$$

$$\mathcal{P}_1 : \{P_1, \vec{n}_1\}$$

$$\mathcal{P}_2 : \{P_2, \vec{n}_2\}$$

它们之间存在 6 种构形，如图 11.24 所示。根据 Dan Sunday 的分类法（Sunday 2001 c），我们可以用相交（或不相交）和描述构形的代数条件来描述每一种构形（如表 11.1 所示）。

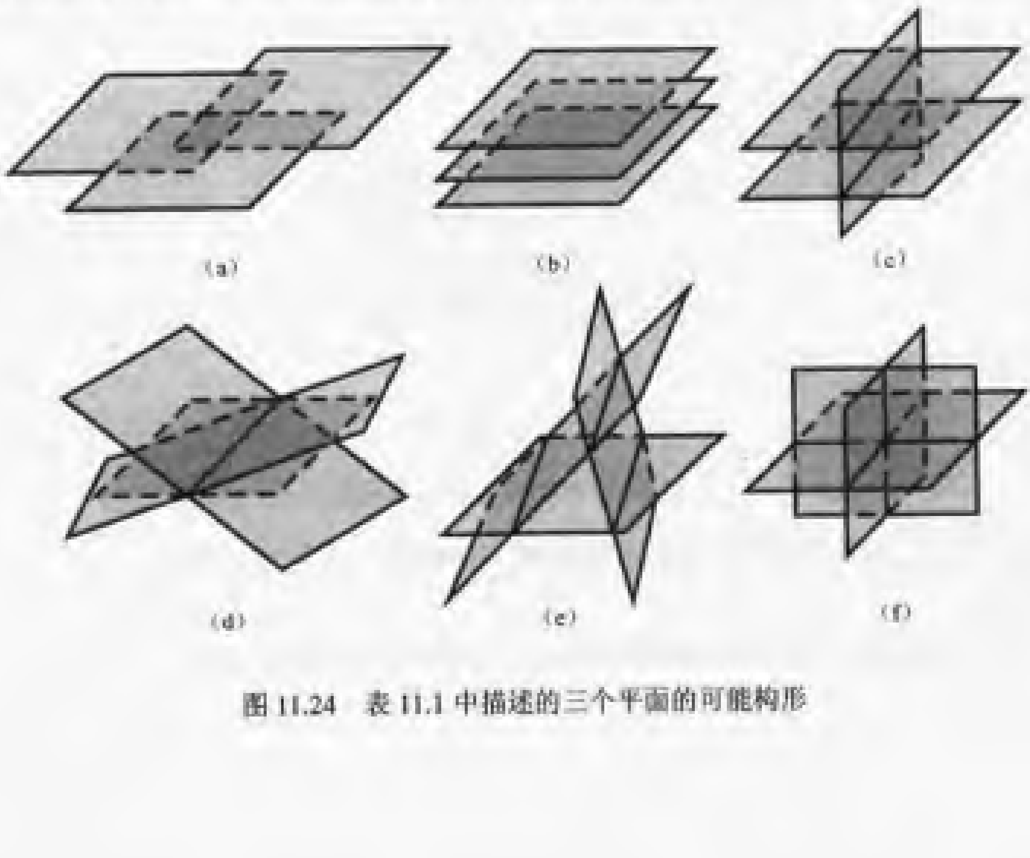


图 11.24　表 11.1 中描述的三个平面的可能构形

表 11.1 可用测试向量代数条件来区分的三个平面的 6 种可能构形

构 形	相交情况	条 件
所有平面都互相平行		$\vec{n}_i \times \vec{n}_j = 0, \forall i, j \in \{0, 1, 2\}$
共面（图 11.24a）	平面	$\vec{n}_0 \cdot P_0 = \vec{n}_1 \cdot P_1 = \vec{n}_2 \cdot P_2$
不相交（图 11.24b）	不存在	$\vec{n}_0 \cdot P_0 \neq \vec{n}_1 \cdot P_1 \neq \vec{n}_2 \cdot P_2$
只有两个平面互相平行（或共面）（图 11.24c）	两条平行直线 （或一条直线）	只有一个 $\vec{n}_i \times \vec{n}_j = 0, \forall i, j \in \{0, 1, 2\}$
不存在两个平面互相平行		$\vec{n}_i \times \vec{n}_j \neq 0, \forall i, j \in \{0, 1, 2\}, i \neq j$
相交直线平行		$\vec{n}_0 \cdot (\vec{n}_1 \times \vec{n}_2) = 0$
相交所得的直线共线（图 11.24d）	一条直线	一条直线上的测试点
不相交（图 11.24e）	三条平行直线	
相交所得的直线不平行（图 11.24f）	点	$\vec{n}_0 \cdot (\vec{n}_1 \times \vec{n}_2) \neq 0$

有必要说明一下：如果任意两个平面$\mathcal{P}_i$和$\mathcal{P}_j$是平行的，那么它们的法线相同，并且可以表示为

$$\vec{n}_i \times \vec{n}_j = 0$$

而且，如果$\mathcal{P}_i$和$\mathcal{P}_j$共面，那么$\mathcal{P}_i$上的任何点都是$\mathcal{P}_j$上的点，这可以表示为

$$\vec{n}_i \cdot P_i = \vec{n}_j \cdot P_j$$

这些条件使我们能区分前 3 种情形，以及将前 3 种情形与其他情形区分开。

如果任何两个平面都不相交，那么将得到构形（d），（e）或（f）中的一种。为了区别（f）与其他两种构形（d）和（e），我们注意到，$\mathcal{P}_1$和$\mathcal{P}_2$必定相交于一条直线，而且这条直线必定与$\mathcal{P}_0$相交于一个点。与我们在两个平面的相交中所看到的问题一样，相交直线的方向与这两个平面法线的叉积方向相同，即$\vec{n}_1 \times \vec{n}_2$。如果$\mathcal{P}_0$与该直线相交于一个点，那么法线$\vec{n}_0$不能与该直线正交。因此，当且仅当满足如下条件时，这三个平面相交

$$\vec{n}_0 \cdot (\vec{n}_1 \times \vec{n}_2) \neq 0$$

最后一个方程可以被看成是数量三元积（参见 3.3.2 节），也就是 3×3 的平面法线系数矩阵的行列式。

Goldman（1990a）将计算交点的公式精简为

$$P = ((P_0 \cdot \vec{n}_0)(\vec{n}_1 \times \vec{n}_2) + (P_1 \cdot \vec{n}_1)(\vec{n}_2 \times \vec{n}_0) + (P_2 \cdot \vec{n}_2)(\vec{n}_0 \times \vec{n}_1))/\vec{n}_0 \cdot (\vec{n}_1 \times \vec{n}_2)$$

如果平面显式表示为

$$\mathcal{P}_0 : a_0 x + b_0 y + c_0 z + d_0 = 0$$

$$\mathcal{P}_1 : a_1 x + b_1 y + c_1 z + d_1 = 0$$

$$\mathcal{P}_2 : a_2 x + b_2 y + c_2 z + d_2 = 0$$

那么我们可以将问题看做求解 3 个同时存在的线性方程。可以使用高斯消元法和克拉默法则（参见 2.7.4 节）之类的技术来求解这类问题。Bowyer 和 Woodwark（1983）将其精简如下：

$$BC = b_1c_2 - b_2c_1$$

$$AC = a_1c_2 - a_2c_1$$

$$AB = a_1b_2 - a_2b_1$$

$$DC = d_1c_2 - d_2c_1$$

$$DB = d_1b_2 - d_2b_1$$

$$AD = a_1d_2 - a_2d_1$$

$$\text{invDet} = \frac{1}{a_0BC - b_0AC - c_0AB}$$

$$X = (b_0DC - d_0BC - c_0DB) * \text{invDet}$$

$$Y = (d_0AC - a_0DC - c_0AD) * \text{invDet}$$

$$Z = (b_0AD - a_0DB - c_0AB) * \text{invDet}$$

11.5.3 三角形与平面的相交

假设我们有一个用一个点和一条法线定义的平面$\mathcal{P}$，以及一个由三个顶点Q_0，Q_1和Q_2所定义的三角形$\mathcal{T}$，如图 11.25 所示。如果平面与三角形相交，那么三角形的一个顶点将与其他的两个顶点分别位于平面的两边。如果我们计算每一个顶点Q_0，Q_1和Q_2与$\mathcal{P}$之间的有符号距离（参见 10.3.1 节），并比较它们的符号，那么，我们就能立即确定三角形与平面是否相交。不失一般性，假设Q_0与Q_1和Q_2位于平面$\mathcal{P}$的两侧，那么两条边Q_0Q_1和Q_0Q_2必定与平面分别相交于点I_0和I_1。线段I_0I_1就是$\mathcal{P}$和$\mathcal{T}$的交线段（参见 11.1.1 节）。

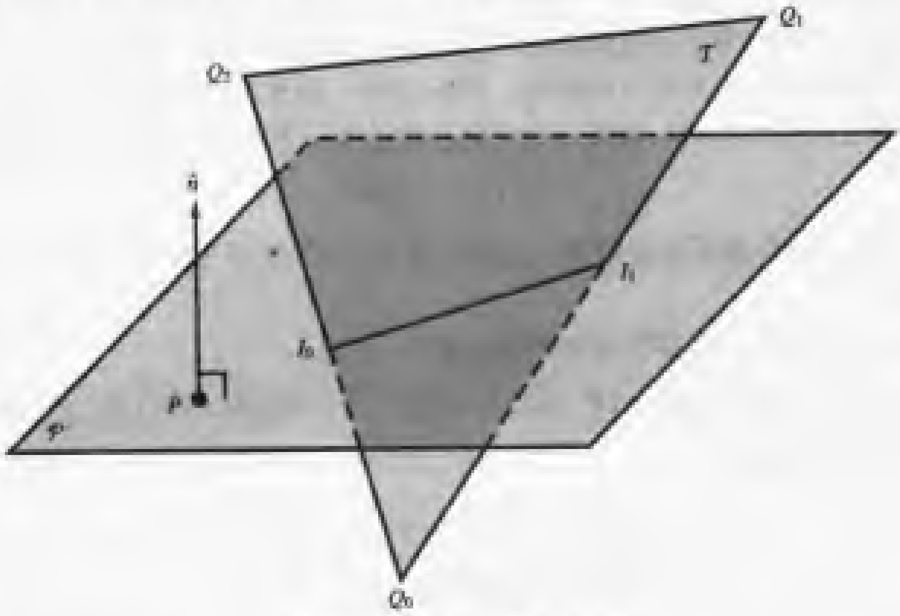


图 11.25　平面－三角形相交

如图 11.26 所示，存在许多的“退化”情形。所有的这类情形都与一个或多个三角形顶点刚好位于平面内有关。在图 11.26（a）中，$\mathcal{P}$和$\mathcal{T}$共面，一般认为这不是一种相交，

虽然应用程序的实际要求必须进行特殊的处理。可用相同的方法来处理图 11.26（b）和图 11.26（c）的情形，即认为位于平面内的Q_i与相交无关。然而，图 11.26（d）中的情形令人更感兴趣，因为“相交”包含一条有限的线段，与“正常的”相交一样。应用程序的实际需要决定如何处理这类情形。

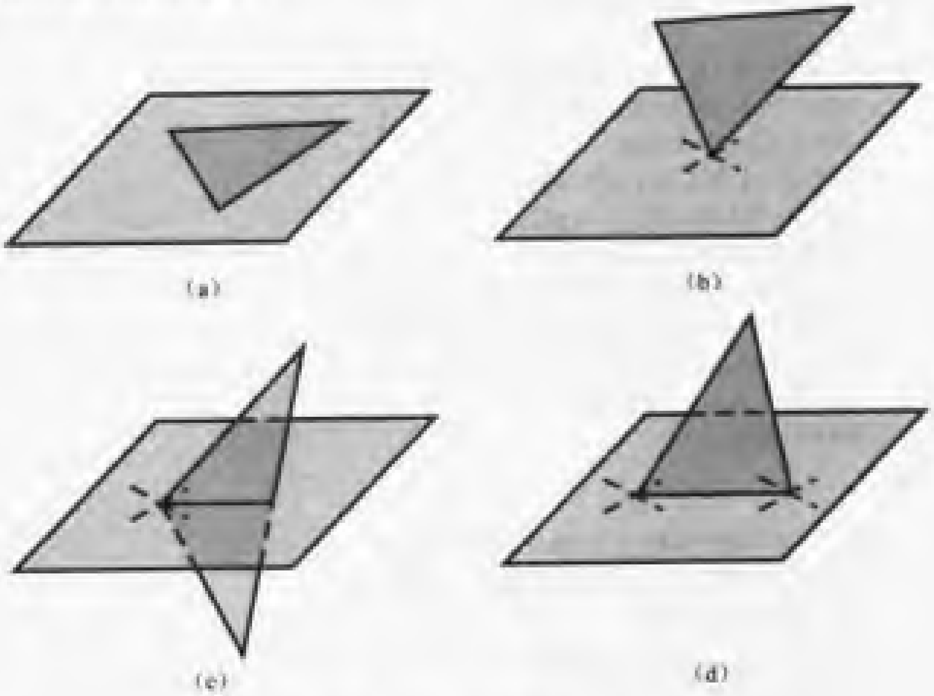


图 11.26　平面—三角形相交的构形

伪码为

```
bool IntersectionOfTriangleAndPlane(Plane3D pl,Triangle3D tri, p2,
                                    Intersection isect)
{
    float dot1, dot2, dot3;
    dot1 = Dot(pl.normal, tri.p1 - pl.pointOnPlane);
    dot2 = Dot(pl.normal, tri.p2 - pl.pointOnPlane);
    dot3 = Dot(pl.normal, tri.p3 - pl.pointOnPlane);

    if (fabs(dot1) <= EPSILON) dot1 = 0.0;
    if (fabs(dot2) <= EPSILON) dot2 = 0.0;
    if (fabs(dot3) <= EPSILON) dot3 = 0.0;
    d1d2 = dot1 * dot2;
    d1d3 = dot1 * dot3;

    if (d1d2 > 0.0 && d1d3 > 0.0) {
        // all points above plane
        return false;
    } else if (d1d2 < 0.0 && d1d3 < 0.0) {
        // all points below plane
        return false;
    }
```

```
    if (fabs(dot1) + fabs(dot2) + fabs(dot3) == 0) {
        // coplanar case
        isect.type = plane;
        return true;
    }
    // Most common intersection

    if ((dot1 > 0 && dot2 > 0 && dot3 < 0) ||
        (dot1 < 0 && dot2 < 0 && dot3 > 0) {
        isect.type = line;
        Line3D l1(tri.p1, tri.p3);
        Line3D l2(tri.p2, tri.p3);
        Point3D point1, point2;
        LineIntersectPlane(plane, l1, point1);
        LineIntersectPlane(plane, l2, point2);
        isect.line.d = point2 - point1;
        isect.line.p = point1;
        return true;
    }

    if ((dot2 > 0 && dot3 > 0 && dot1 < 0) ||
        (dot2 < 0 && dot3 < 0 && dot1 > 0) {
        isect.type = line;
        Line3D l1(tri.p2, tri.p1);
        Line3D l2(tri.p3, tri.p1);
        Point3D point1, point2;
        LineIntersectPlane(plane, l1, point1);
        LineIntersectPlane(plane, l2, point2);
        isect.line.d = point2 - point1;
        isect.line.p = point1;
        return true;
    }

    if ((dot1 > 0 && dot3 > 0 && dot2 < 0) ||
        (dot1 < 0 && dot3 < 0 && dot2 > 0) {
        isect.type = line;
        Line3D l1(tri.p1, tri.p2);
        Line3D l2(tri.p3, tri.p2);
        Point3D point1, point2;
        LineIntersectPlane(plane, l1, point1);
        LineIntersectPlane(plane, l2, point2);
        isect.line.d = point2 - point1;
        isect.line.p = point1;
        return true;
    }

    // Case b
    if (dot1 == 0 && ((dot2 > 0 && dot3 > 0) || (dot2 < 0 && dot3 < 0))) {
        isect.type = point;
```

```
        isect.point = tri.p1;
        return true;
    }

    if (dot2 == 0 && ((dot1 > 0 && dot3 > 0) || (dot1 < 0 && dot3 < 0))) {
        isect.type = point;
        isect.point = tri.p2;
        return true;
    }

    if (dot3 == 0 && ((dot2 > 0 && dot1 > 0) || (dot2 < 0 && dot1 < 0))) {
        isect.type = point;
        isect.point = tri.p3;
        return true;
    }

    // Case c
    if (dot1 == 0 && ((dot2 > 0 && dot3 < 0) || (dot2 < 0 && dot3 > 0))) {
        isect.type = line;
        Line3D l1(tri.p3, tri.p2);
        Point3D point1;
        LineIntersectPlane(plane,l1, point1);
        isect.line.d = point1 - tri.p1;
        isect.line.p = tri.p1;
        return true;
    }

    if (dot2 == 0 && ((dot1 > 0 && dot3 < 0) || (dot1 < 0 && dot3 > 0))) {
        isect.type = line;
        Line3D l1(tri.p1, tri.p3);
        Point3D point1;
        LineIntersectPlane(plane,l1, point1);
        isect.line.d = point1 - tri.p2;
        isect.line.p = tri.p2;
        return true;
    }

    if (dot3 == 0 && ((dot1 > 0 && dot2 < 0) || (dot1 < 0 && dot2 > 0))) {
        isect.type = line;
        Line3D l1(tri.p1, tri.p2);
        Point3D point1;
        LineIntersectPlane(plane, l1, point1);
        isect.line.d = point1 - tri.p3;
        isect.line.p = tri.p3;
        return true;
    }

    // Case d
    if (dot1 == 0 && dot2 == 0) {
        isect.type = line;
```

```
        isect.line.d = tri.p2 - tri.p1;
        isect.line.p = tri.p1;
        return true;
    }

    if (dot2 == 0 && dot3 == 0) {
        isect.type = line;
        isect.line.d = tri.p3 - tri.p2;
        isect.line.p = tri.p2;
        return true;
    }

    if (dot1 == 0 && dot3 == 0) {
        isect.type = line;
        isect.line.d = tri.p3 - tri.p1;
        isect.line.p = tri.p1;
        return true;
    }

    return false;
}
```

11.5.4　三角形与三角形的相交

在本节中，我们将讨论两个三角形的相交问题。为了便于本节的讨论，我们将两个三角形分别定义为三个顶点的集合：

$$\mathcal{T}_0 : \{V_{0,0}, V_{0,1}, V_{0,2}\}$$

$$\mathcal{T}_1 : \{V_{1,0}, V_{1,1}, V_{1,2}\}$$

存在许多的不同构形，三角形一三角形相交方法必须处理它们（如图 11.27 所示）：平面$\mathcal{P}_0$与$\mathcal{P}_1$可能平行且共面、平行不共面或不平行，同时$\mathcal{T}_0$与$\mathcal{T}_1$可能相交或不相交。不论哪种算法，都必须将共面的构形作为特殊情形来处理（稍候我们将进行更多的讨论），相似的情形是三角形本身退化的情形（两个或多个顶点共面）。

最直接的算法就是简单地测试每一个三角形的每一条边与另一个三角形（面）是否相交：当找到第一个边一面相交时，返回真；如果没有找到边一面相交，返回假。如果有一个高效的线段一三角形相交函数，这种算法并不算差。然而，其他的几种算法要快得多。

Möller 和 Haines（Möller 和 Haines 1999；Möller 1997）描述了一种确定两个三角形相交的“区间重叠方法”。如果它们相交，那么可以直接得到相交的线段。这种方法的理论基础是：如果我们已经排除了顶点完全位于对方平面两侧的三角形对，那么这两个平面的相交直线$\mathcal{L}$与两个三角形将相交。两个平面的相交直线$\mathcal{L}$被每一个三角形分别“剪切”成两条线段（“区间”）。如果这两条线段重叠，那么三角形相交；否则，它们不相交。可以简单地计算出直线$\mathcal{L}$（参见 11.5.1 节），假设有直线

$$\mathcal{L}(t) = P + t\vec{d}$$

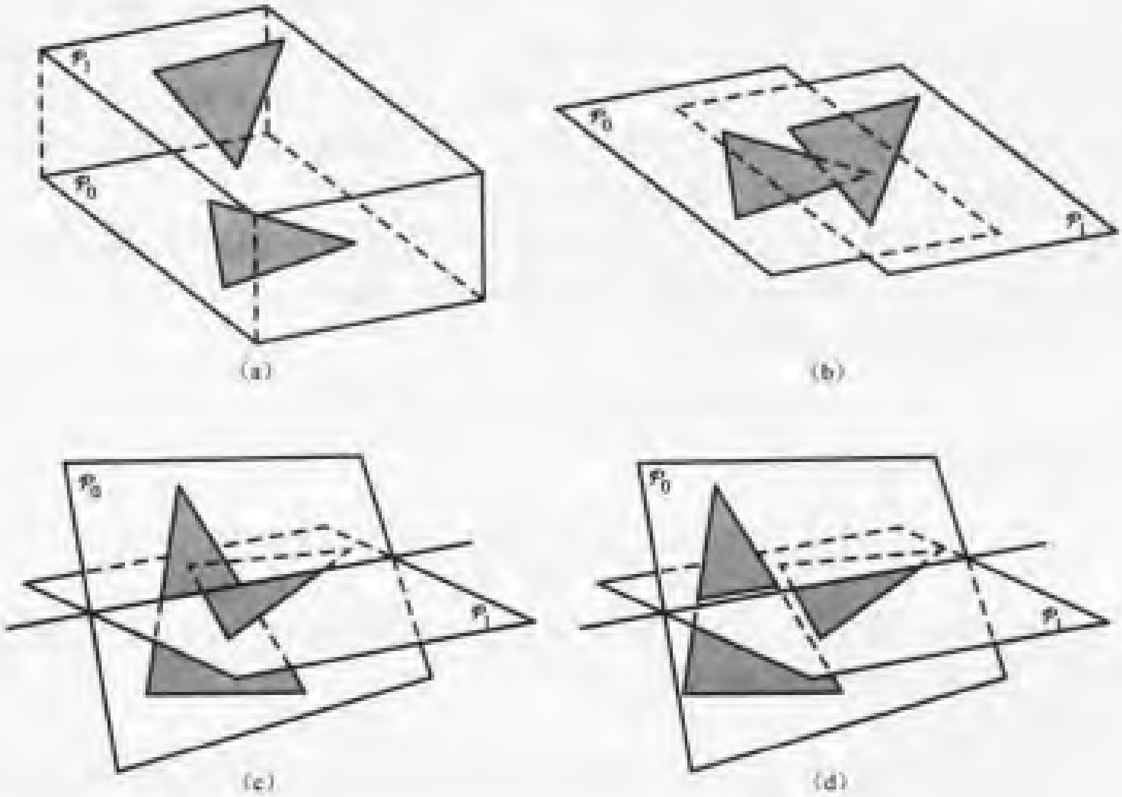


图 11.27　三角形—三角形相交的构形：(a) $\mathcal{P}_0 \| \mathcal{P}_1$，但$\mathcal{P}_0 \neq \mathcal{P}_1$；(b) $\mathcal{P}_0 = \mathcal{P}_1$；(c) $\mathcal{T}_0$与$\mathcal{T}_1$相交；(d) $\mathcal{T}_0$与$\mathcal{T}_1$不相交

设$t_{0,0}$和$t_{0,1}$为$\mathcal{L}$上描述与$\mathcal{T}_0$相交的线段的参数值，而$t_{1,0}$和$t_{1,1}$为$\mathcal{L}$上描述与$\mathcal{T}_1$相交的线段的参数值。图 11.28 显示了区间之间的各种可能关系。很清楚，我们可以很容易地排除不相交的三角形（如图所示，当区间毗邻时，你可以做出“策略性”决定），只有在检测到相交存在且希望知道比是否相交更多的信息时，才计算确切的相交。

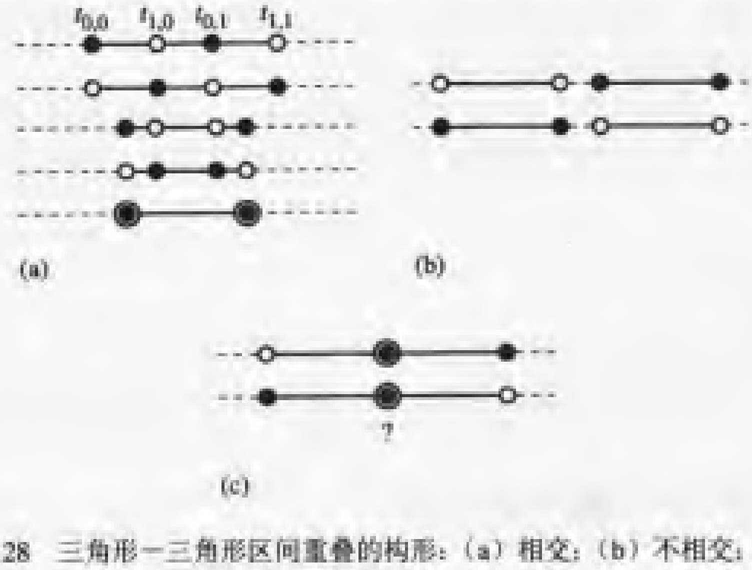


图 11.28　三角形—三角形区间重叠的构形：(a) 相交；(b) 不相交；(c) ？

为了快速地排除不可能相交的三角形，我们已经在前面假设测试了每一个三角形的顶

点相对另一个三角形所在的平面的位置关系。区间重叠方法通过检测每一个点与另一个三角形所在的平面之间的有符号距离来实现这种排除（参见 10.3.1 节）。现在可以再次利用计算得到的有符号距离：我们知道每一个三角形的一个顶点与另两个顶点分别位于$\mathcal{L}$的两侧（三角形的一个或两个顶点有时可能刚好位于$\mathcal{L}$上，但只需在实现中增加一点点代码就能处理这种情形）。不失一般性，假设$V_{0,0}$和$V_{0,1}$位于$\mathcal{L}$的同一侧。

现在，只需简单地计算两条三维直线的交点，我们就能得出边$V_{0,0}V_{0,2}$和$V_{0,1}V_{0,2}$在$\mathcal{L}$上的交点。然而，Möller 和 Haines 采用了一种更聪明的优化方法。

（1）将三角形顶点$V_{0,i}$投影到$\mathcal{L}$上：

$$V'_{0,i} = \vec{d} \cdot (V_{0,i} - P), \qquad i \in \{0, 1, 2\}$$

（2）计算$t_{0,0}$和$t_{0,1}$如下：

$$t_{0,i} = V'_{0,i} + (V'_{0,2} - V'_{0,i}) \frac{\text{dist}_{V_{0,i}}}{\text{dist}_{V_{0,i}} - \text{dist}_{V_{0,2}}}, \qquad i \in \{0, 1\}$$

如果实际的相交就是我们要计算的，那么我们可以通过考察$t_{0,0}$, $t_{0,1}$, $t_{1,0}$和$t_{1,1}$来找到重叠的区间，然后将参数值代回$\mathcal{L}$的方程中。

整个算法的要点如下。

（1）确定$\mathcal{T}_0$或$\mathcal{T}_1$之一（或同时）是否退化，并根据应用程序的实际需要来进行处理（这可能是退出应用程序或继续进行）。

（2）计算$\mathcal{T}_0$的平面方程。

（3）计算$\mathcal{T}_1$的顶点的有符号距离$\text{dist}_{V_{1,i}}$，$i \in \{0, 1, 2\}$。

（4）比较$\text{dist}_{V_{1,i}}$，$i \in \{0, 1, 2\}$的符号：如果它们的符号都相同，则返回假；否则，处理下一步。

（5）计算$\mathcal{T}_1$的平面方程。

（6）如果$\mathcal{P}_0$或$\mathcal{P}_1$的平面方程相同（或在误差范围之内），那么比较它们的d值，以确定它们是否共面：

- 如果共面，则将三角形投影到与三角形的平面最接近同向的轴对齐带中，并执行二维空间的三角形相交测试。
- 否则，平行的平面不共面，因此不可能相交，退出应用程序。

（7）比较$\text{dist}_{V_{0,i}}$，$i \in \{0, 1, 2\}$的符号：如果它们的符号相同，则返回假；否则，继续处理下一步。

（8）计算相交直线。

（9）计算区间：

- 如果区间不重叠，则三角形不相交。返回假。
- 否则，如果需要求相交的线段，就计算该线段。任何情形下都返回真。

11.6　平面对象与多面体的相交

在本节中，我们将讨论平面对象与多面体之间的相交，多面体是多边形网格的一种特

殊情形。多边形网格是满足如下条件的顶点、边和面的组合：

i．每一个顶点必须共享至少一条边（不允许存在孤立的顶点）。

ii．每一条边必须共享至少一个面（不允许存在孤立的边或折线）。

iii．如果两个面相交，那么相交的顶点或边必须是网格上的一个成分（不允许存在贯通的面。一个面的一条边不能位于另一个面之内）。

面是位于三维空间中的凸多边形。许多应用程序的存储方式很简单，并且对面的操作太简单，它们仅仅支持三角形的面。也允许存在非凸多边形的面，这样将使实现和对对象的操作复杂化。如果所有的面都是三角形的，那么，该对象就叫做三角形网格，或简称三角网格。

多面体是满足额外的限制条件的多边形网格。在直观上，多面体包围一个封闭的空间区域，并且没有不必要的边连接。最简单的例子是四面体，一种具有 4 个顶点、6 条边和 4 个三角形面的多边形网格。附加的限制条件包括：

- 当被看成是节点为面而且弧为相邻的面所共享的边的图形时，网格是连通的。直观上，如果你能沿着从超始面到目的面的成对的相邻面的路径，从任何起始面都能到达目的面，那么该网格就是连通的。
- 每一条边都刚好被两个面所共享。这个条件确保网格是一个闭合且有界的表面。

11.6.1 三角网格

如果多面体的面都是三角形，那么多面体与平面或三角形相交的问题就变得相对简单了：我们可以简单地对多面体的每一个三角形面运用三角形一平面相交算法（参见 11.5.3 节）或三角形一三角形相交算法（参见 11.5.4 节）。如果注意到许多与三角形网格的顶点有关的运算（例如，检查顶点位于平面的哪一面）都只需进行一次，就能得到一些高效的算法。

一般地，三角形网格与一个平面或三角形的相交包含顶点、边和折线（封闭或开放）的组合。图 11.29 显示了一个三角网格（在这种情形中，是一个四面体）与一个平面的相交。粗的折线表示相交，其中包含三条线段，它们连在一起形成一条折线。

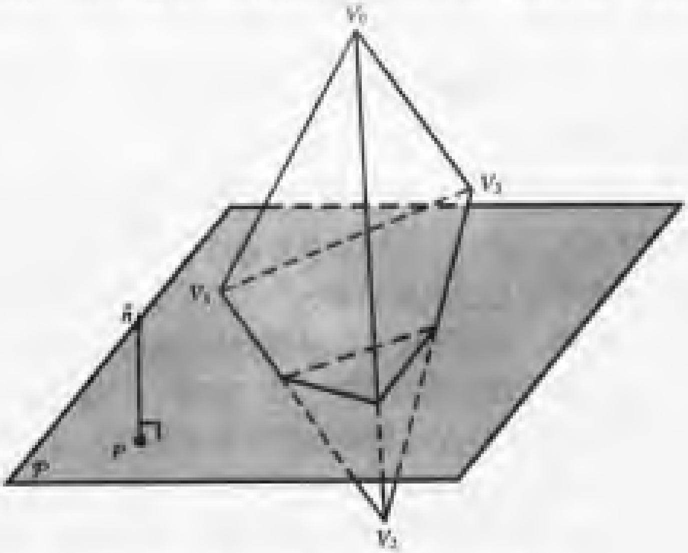


图 11.29 三角网格与平面的相交

11.6.2　一般多面体

在我们的定义中，多面体可以具有边为任意数目的面，只需满足面是凸的这一限制。如果我们要计算这样的多面体与平面或三角形的相交，那么就必须解决多边形与平面或三角形的相交问题。如果我们应用与前面描述的处理三角网格与平面或三角形相交相同的方法，那么我们也将得到顶点、边和折线的组合。

1. 平面与多边形的相交

图 11.30 显示了一个凸多边形与一个平面的相交，边V_2V_3和V_4V_5与平面相交，而且相交包含一条直线（显示为粗体）。解决这一问题的方法是计算三角形与平面相交的简单方法（参见 11.5.3 节）：如果平面与包含多边形的平面不平行，那么它们可能相交。我们计算每一个点$V_0, V_1, \ldots, V_{n-1}$与平面的有符号距离，如果符号不相同，那么存在相交。如果它的两个顶点有不同的符号，那么边V_iV_{i+1}与平面相交；如果存在相交，则自然存在两条这样的边。然而，如果有一个或多个顶点刚好位于平面上，就应该小心处理。

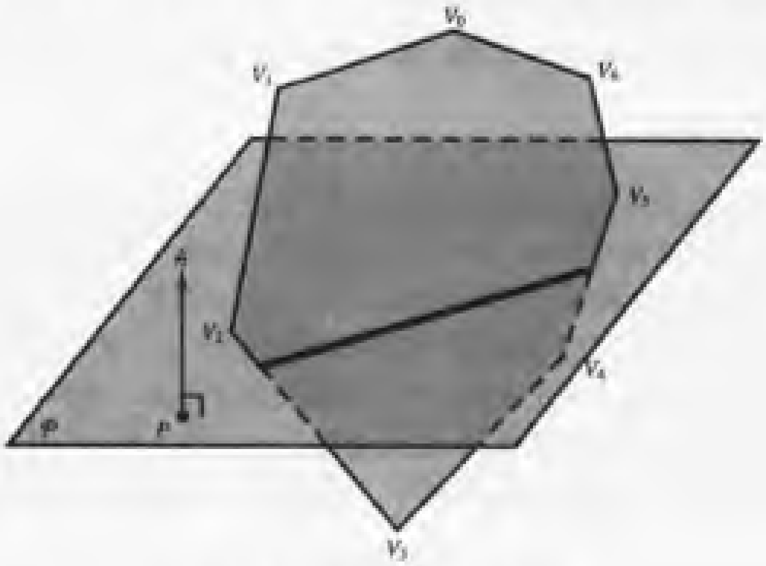


图 11.30　多边形与平面的相交

2. 三角形与多边形的相交

图 11.31 显示了一个凸多边形与一个三角形的相交。我们也可以首先检查多边形所在的平面与三角形所在的平面是否平行，以确定是否存在相交。如果可能存在相交，那么我们也可以计算多边形所在的平面与三角形的每一个顶点之间的有符号距离，以确定三角形是否穿过多边形的平面。然而，这并不能说明多边形与平面是否相交。我们还必须检查三角形和多边形的边，以确定是否有任何的边穿过对方的面。注意到如下的事实可以将这一方法简化一点，即我们只需检测边V_iV_{i+1}的顶点符号是否相同。当我们相对三角形检测多边形的边时，我们可以简单地利用 11.1.2 节描述的方法，以确定边是否相交于三角形的内部。然而，为了检测边Q_iQ_{i+1}是否相交于多边形的内部，我们必须使用较低效的方法，即 11.1.3 节中描述的计算线段／多边形相交的方法。

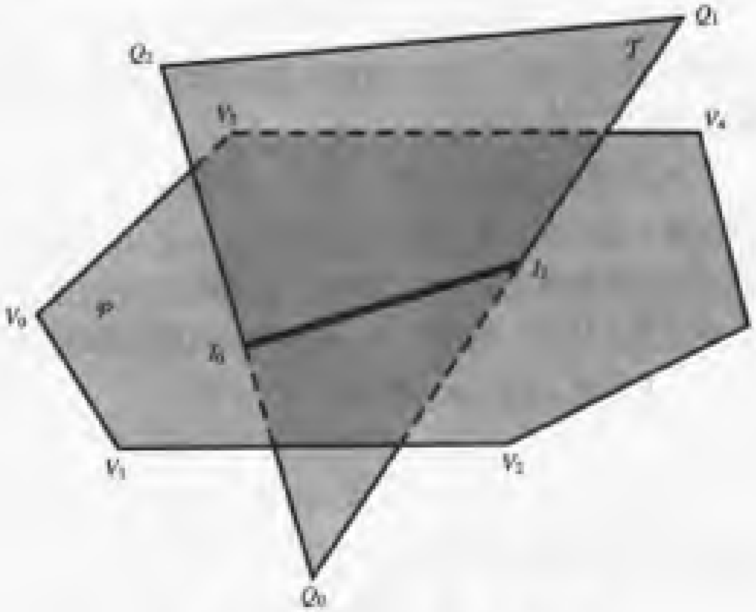


图 11.31 多边形与三角形的相交

3. 平面对象与多面体的相交

为了确定平面对象与多面体的相交，我们只需简单地对多面体的每一个面应用上述的三角形－多边形或平面－多边形相交的方法。再次注意，在测试每一个面之前先计算所有顶点的有符号距离，在测试每一个面时，具有相同符号的距离就不需重复计算，这样能提高一些效率。

11.7 平面对象与二次曲面的相交

我们在本节将全面介绍平面对象与二次曲线的相交。我们提供了多种方法——一种针对一般形式的平面与任何二次曲线相交的方法，一种针对平面与所谓的“自然二次曲线”（球面、正圆锥面、正圆柱面）之间相交的方法，以及一种针对三角形与圆锥面相交的方法。

11.7.1 平面与一般二次曲面的相交

在本节中，我们讨论计算平面与一般二次曲面之间相交的问题（Busboom 和 Schalkoff 1996）。这里介绍的方法可用于所有的二次曲面，但是由于这是一种通用的方法，因而它不能利用特殊类型的二次曲面的几何性质。我们在几个子节中分别介绍处理一些特殊二次曲面的方法。

一般的二次曲面可以描述为

$$ax^2+by^2+cz^2+2fyz+2gzx+2hyx+2px+2qy+2rz+d=0 \qquad (11.17)$$

系数a,b,c,d,e,f,g,h,p,q和r的不同值决定二次曲面的不同类型（参见9.4节）。

为了解决这一问题，我们通过指定平面上的一个点P和平面上的两个正交向量$\vec{u}$和$\vec{v}$来定义平面：

$$\mathcal{P}(t_u, t_v) = P + t_u\vec{u} + t_v\vec{v} \tag{11.18}$$

通过将方程（11.18）代入方程（11.17）就能求得该平面与上述二次曲面之间的相交。这样得到一个用 t_u 和 t_v 表示的二次方程：

$$At_u^2 + Bt_ut_v + Ct_v^2 + Dt_u + Et_v + F = 0 \tag{11.19}$$

于是求解相交的问题就相当于计算上述方程的系数 A, B, C, D, E 和 F。

为了不处理如此庞大的代数表达式，我们采用一种稍微不同的方法：如果我们用矩阵来表示方程（11.17）和方程（11.19）的系数集，那么我们就能将变换表示为一个矩阵。设

$$\mathbf{C} = [A \quad B \quad C \quad D \quad E \quad F]^{\mathrm{T}}$$

以及

$$\mathbf{Q} = [a \quad b \quad c \quad f \quad g \quad h \quad p \quad q \quad r \quad d]^{\mathrm{T}}$$

于是，我们的变换就是满足下式的矩阵 **M**

$$\mathbf{C} = \mathbf{MQ}$$

其中

$$\mathbf{M} = \begin{bmatrix} u_x^2 & u_y^2 & u_z^2 & 2u_yu_z & 2u_zu_x & 2u_xu_y & 0 & 0 & 0 & 0 \\ 2u_xv_x & 2u_yv_y & 2u_zv_z & 2(u_yv_z + v_yu_z) & 2(u_zv_x + v_zu_x) & 2(u_xv_y + v_xu_y) & 0 & 0 & 0 & 0 \\ v_x^2 & v_y^2 & v_z^2 & 2v_yv_z & 2v_zv_x & 2v_xv_y & 0 & 0 & 0 & 0 \\ 2P_xu_x & 2P_yu_y & 2P_zu_z & 2(P_yu_z + u_yP_z) & 2(P_zu_x + u_zP_x) & 2(P_xu_y + u_xP_y) & 2u_x & 2u_y & 2u_z & 0 \\ 2P_xv_x & 2P_yv_y & 2P_zv_z & 2(P_yv_z + v_yP_z) & 2(P_zv_x + v_zP_x) & 2(P_xv_y + v_xP_y) & 2v_x & 2v_y & 2v_z & 0 \\ P_x^2 & P_y^2 & P_z^2 & 2P_yP_z & 2P_zP_x & 2P_xP_y & 2P_x & 2P_y & 2P_z & 1 \end{bmatrix}$$

11.7.2　平面与球面的相交

在本节中，我们讨论计算平面与球面相交的问题，如图 11.32 所示。

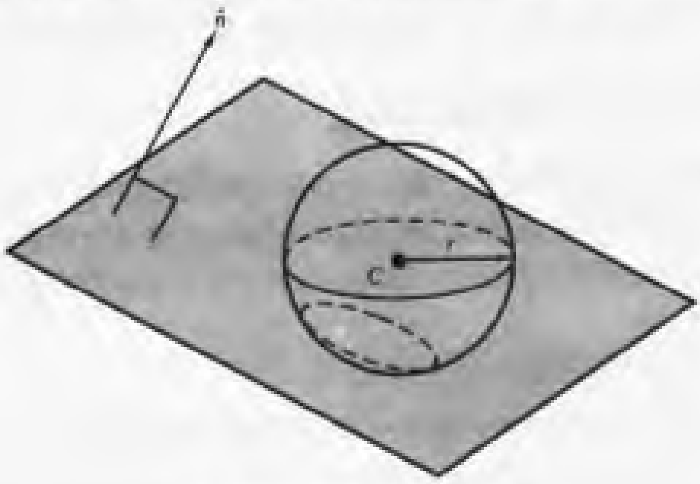


图 11.32　平面与球面的相交

球面可以隐式定义为

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 - r = 0 \qquad (11.20)$$

其中C为球面的中心，r是半径。如果我们用参数形式来定义平面，就像我们在9.2.1节中所做的一样，即：

$$\mathcal{P}(t_u, t_v) = P + t_u\vec{u} + t_v\vec{v}$$

可以将该方程代入方程（11.20），可得到一个形式如下的二次曲线

$$At_u^2 + Bt_ut_v + Ct_v^2 + Dt_u + Et_v + F = 0$$

如果进行代入、展开及合并同类项，可得

$$A = u_x^2 + u_y^2 + u_z^2$$

$$B = 2(u_xv_x + u_yv_y + u_zv_z)$$

$$C = v_x^2 + v_y^2 + v_z^2$$

$$D = 2(P_xu_x + P_yu_y + P_xu_z) - 2(C_xv_x + C_yv_y + C_xv_z)$$

$$E = 2(P_xv_x + P_yv_y + P_xv_z) - 2(C_xv_x + C_yv_y + C_xv_z)$$

$$F = C_x^2 + C_y^2 + C_z^2 + P_x^2 + P_y^2 + PC_z^2 - 2(C_xP_x + C_yP_y + C_zP_z) - r$$

另一种方法是利用更直接的几何方法。很清楚，如果平面和球面相交，那么相交的结果就是平面上的一个圆。平面的方程提供了三维空间的一个圆的部分定义，因此我们需要计算圆的圆心和半径。注意到球面的中心C位于一条通过相交圆Q的中心并且为平面$\mathcal{P}$的法线的直线上，就能深入理解这一方法。从图11.33显示的截面中，可以更直接地看出这一点。如果我们的平面是用正规化的非坐标形式来隐式表示的，即

$$P \cdot \hat{n} + d = 0$$

那么，该平面与球面的中心之间的距离可以简单地表示为

$$b = \hat{n} \cdot Q + d$$

（参见10.3.1节）。如果$|b| > r$，那么它们不相交；否则圆Q的圆心就是

$$Q = C - b\hat{n}$$

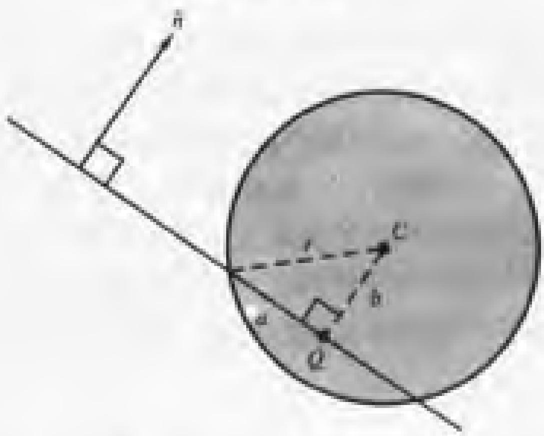


图11.33 球面一平面相交的横截面视图

我们现在还需要做的就是确定相交圆的半径。再观察图 11.33，我们可以很容易地看出

$$a^2+b^2=r^2$$

因而

$$a=\sqrt{r^2-b^2}$$

就是要求的半径。

如果平面刚好“擦过”球面，那么r^2-b^2将是一个非常小的数。在这种情形中，应用程序可能希望相交实际上只是一个点，并进行相应的处理。

伪码为

```
bool PlaneSphereIntersection(Plane3D plane, Sphere sphere, Intersection isect)
{
    Vec3D v1 = sphere.center - plane.pointOnPlane;
    // normal is unit length
    float b = fabs(dotProd(plane.normal,v1));
    if (b < r) {
        Point3D Q = sphere.center - b * plane.normal;
        float radius = sqrt(sphere.radius^2 + b^2);
        if (radius < epsilon) {
            // consider it as a point
            isect.point = Q;
            isect.type = point;
        } else {
            isect.center = Q;
            isect.radius = radius;
            isect.type = circle;
        }

        return true;
    }
    return false;
}
```

11.7.3 平面与圆柱面的相交

在本节中，我们将讨论计算平面与圆柱面相交的问题，如图 11.34 所示。实际上，圆柱面与平面之间的相交方式有许多种，图 11.35 中显示了其中的 6 种。注意这些相交都显示为是相对于有限的圆柱面的。无限的圆柱面具有较少的相交构形（我们不需要处理端面）；可能的相交是一个圆、一个椭圆、一条直线或一对直线。我们给出了平面与无限圆柱面、平面与有限圆柱面之间的相交的检测算法。前者更简单一些。在下一节中，我们将给出一种计算平面与无限圆柱面的相交的算法。通过用包含端面的平面来“裁剪”相交的二次曲线或直线的范围，可以将这类算法进行扩展，用来处理有限圆柱面。

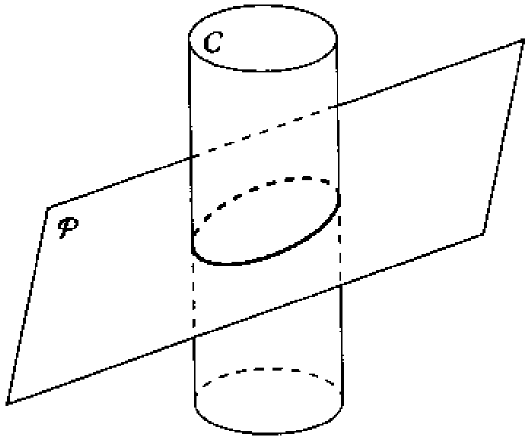


图 11.34　平面与圆柱面的相交

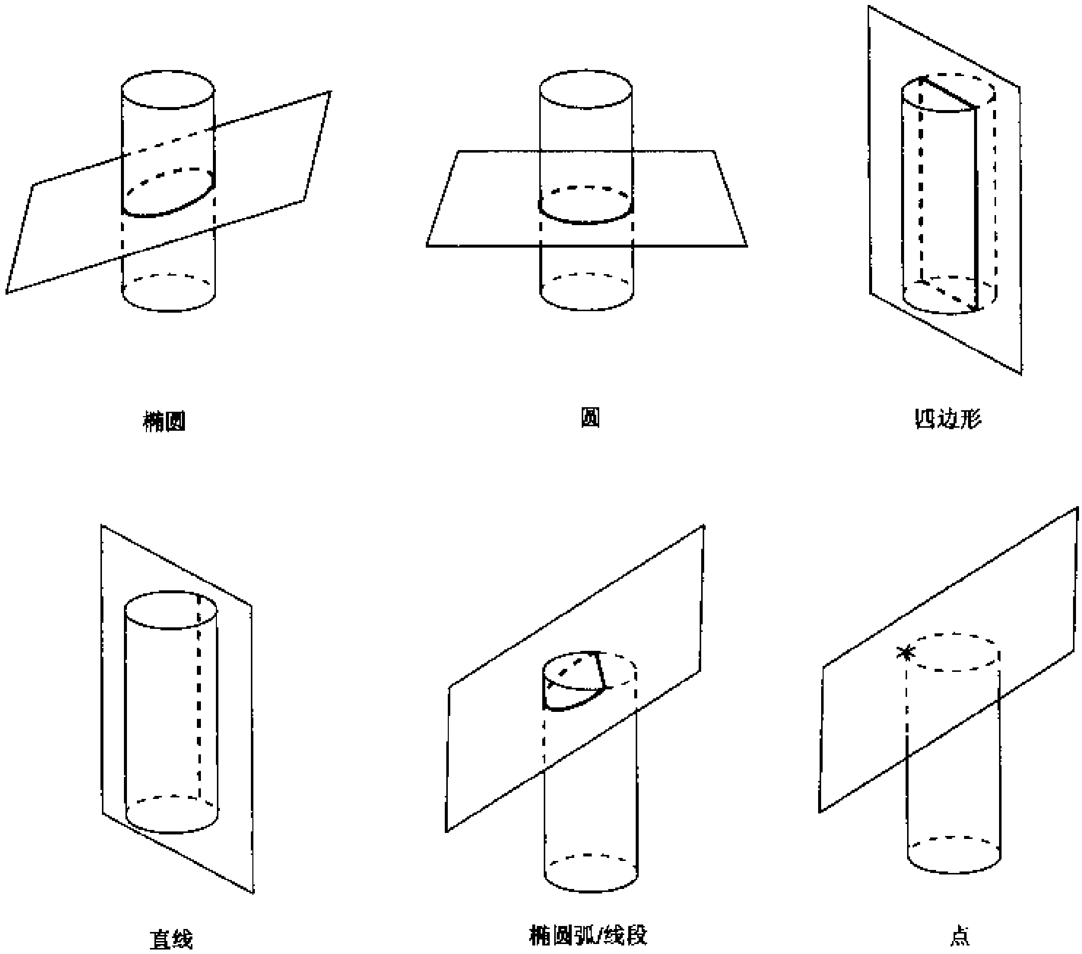


图 11.35　平面与圆柱面相交的几种方式

1．相交检测

为了描述相交检测算法，我们用隐含形式来定义平面：

$$P \cdot \hat{n} + d = 0$$

用坐标来表示的形式一般为 $ax+by+cz+d=0$，其中 $\hat{n}=[a\quad b\quad c]$，而且 $\sqrt{a^2+b^2+c^2}=1$。圆柱面定义于“一般位置”——用中心 C，轴 $\hat{d}$ 和半高 h 来表示（参见 11.3.4 节中的图 11.12）。

（1）无限圆柱面

基于检测相交的目的，平面 $\mathcal{P}$ 和圆柱面 C 可以是如下几种构形中的一种。

① 如果 $\mathcal{P}$ 平行于 C 的轴，则如果 $\mathcal{P}$ 和 C 之间的距离小于或等于圆柱面的半径，那么它们相交。

② 如果 $\mathcal{P}$ 不平行于 C 的轴，$\hat{d}\cdot\hat{n}=1$，那么它们总是相交的。

（2）有限圆柱面

基于检测相交的目的，平面 $\mathcal{P}$ 和圆柱面 C 可以是如下几种构形中的一种。

① $\mathcal{P}$ 可能平行于 C 的轴：$|\hat{d}\cdot\hat{n}|=1$。

② $\mathcal{P}$ 可能垂直于 C 的轴：$\hat{d}\cdot\hat{n}=0$。

③ $\mathcal{P}$ 与 C 的轴可能既不平行也不垂直。在这种情形中，$\mathcal{P}$ 与 C 不一定相交。

我们分别考虑这些情形。

① 如果 $\mathcal{P}$ 平行于 C 的轴，那么，如果 $\mathcal{P}$ 与 C 的轴之间的距离小于或等于圆柱面的半径（此时相交为一条二次曲线或一条直线），则它们相交。

② 如果 $\mathcal{P}$ 垂直于 C 的轴，那么，如果 $\mathcal{P}$ 与 C 之间的距离小于或等于圆柱面的半高（此时相交为一个圆），则它们相交。

③ 如果 $\mathcal{P}$ 与 C 的轴既不平行也不垂直，那么需要考虑两种情形：

a. $\mathcal{P}$ 与 C 的轴的相交图形与 C 的中心之间的距离小于半高，则此时它们一定相交。

b. $\mathcal{P}$ 与 C 的轴的相交图形位于端面外，此时不一定相交，是否相交取决于交点的相对位置及平面与轴之间的夹角。

在每一种情形中，可能相交为一个椭圆、一条椭圆弧和一条直线、或者两条椭圆弧和两条直线，具体情况取决于平面的相对方向。

除最后一种情形外，其他所有情形都很简单。确定平面是否平行于或垂直于圆柱面的轴仅用一次点积就能解决。计算圆柱面的中心 C 与平面之间的距离也很简单和不费时（参见 10.3.1 节）。计算平面与圆柱面的轴的相交也不费时（参见 11.1.1 节）。只有最后一种情形涉及费时的操作，因此必须根据上面讨论的次序来解决。

最后一种情形如图 11.36 所示。显示为横截图并不仅仅是为了图示的方便——确定它们是否相交的方法是在一个通过 C 且与 $\mathcal{P}$ 垂直的平面上进行的。其余的都是简单的三角几何。

如果 I_a 是 $\mathcal{P}$ 与圆柱面的轴的相交图形，那么如果点 I_c 与轴之间的距离小于 r（圆柱面的半径），则存在相交。垂直于 $\mathcal{P}$ 且通过 C 的平面 $\mathcal{P}^{\perp}$ 平行于 $\hat{d}$，因此其法线为

$$\hat{n}\times\hat{d}$$

我们定义 $\mathcal{P}^{\perp}$ 上垂直于 $\hat{d}$ 的一个向量为：

$$\hat{w}=\hat{d}\times(\hat{n}\times\hat{d})$$

$\hat{n}$ 与 $\hat{w}$ 之间的夹角 θ 为

$$\cos(\theta)=\hat{n}\cdot\vec{w}$$

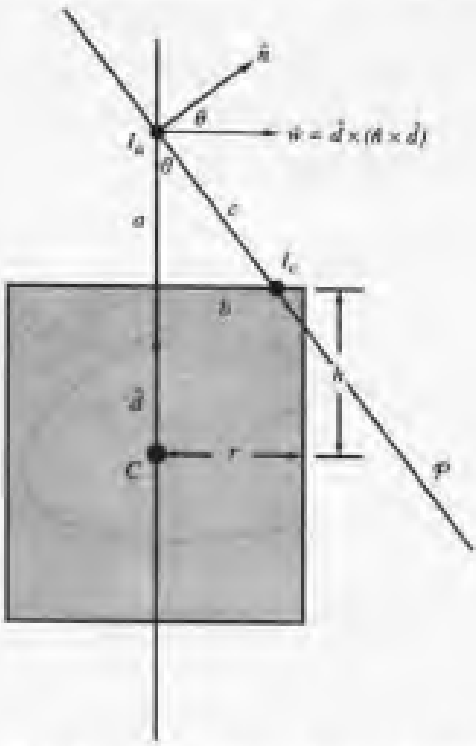


图 11.36　平面－圆柱面相交的边缘视图

我们还知道距离 a 为：

$$a = \|I_a - C\| - h$$

根据余弦函数的定义，我们知道

$$\cos(\theta) = \frac{a}{c}$$

代入可得

$$\hat{n} \cdot \vec{w} = \frac{\|I_a - C\| - h}{c}$$

因此有

$$c = \frac{\|I_a - C\| - h}{\hat{n} \cdot \vec{w}}$$

调用毕达哥拉斯定理（译者注：即勾股定理），可得

$$a^2 + b^2 = c^2$$

$$(\|I_a - C\| - h)^2 + b^2 = \left(\frac{\|I_a - C\| - h}{\hat{n} \cdot \vec{w}}\right)^2$$

$$b^2 = \left(\frac{\|I_a - C\| - h}{\hat{n} \cdot \vec{w}}\right)^2 - (\|I_a - C\| - h)^2$$

因此，如果 $b^2 \leqslant r^2$，则存在相交；否则不存在相交。

2．与无限圆柱面的相交

11.7 节中给出的一般平面－二次曲面的相交公式产生一个用平面来表示的二次曲线的

隐式方程。这种表示方法并不是很方便。对于平面与圆柱面的相交来说，我们可以得到一个可用更直接的几何形式来表示的椭圆或圆；我们的意思是，一个椭圆应该用一个中心C，主轴$\hat{u}$和次轴$\hat{v}$，以及主半径r_u和次半径r_v来定义，如图 11.37 所示，而一个圆应该用圆心C，平面的法线$\hat{n}$和半径r来定义，如图 11.38 所示。

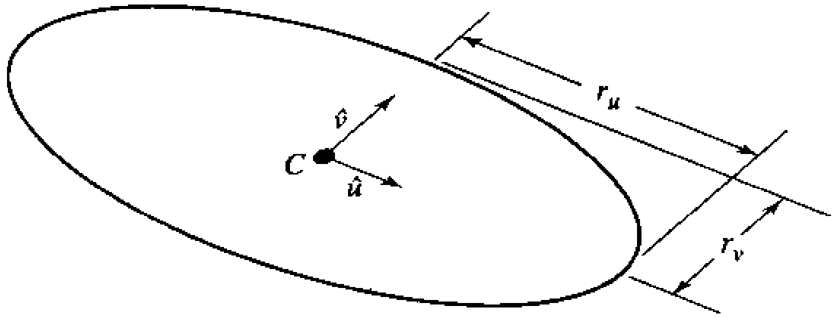


图 11.37　三维空间中的椭圆

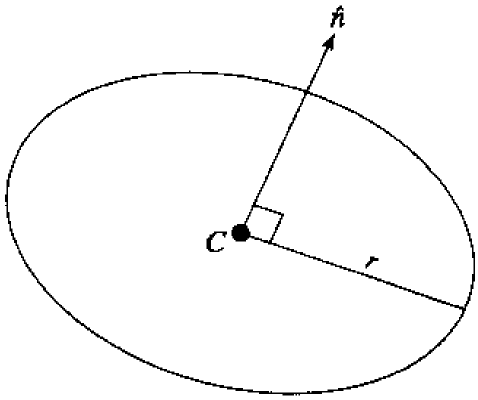


图 11.38　三维空间中的圆

正如在图 11.35 中所看到的一样，一个平面与一个无限圆柱面可能相交为一条直线、两条直线、一个圆或一个椭圆。前面的三种是特殊情形，只有当平面与圆柱面相互之间的角度为一个或两个特殊角度时才会出现，因此，椭圆可以看成是一般的情形。

Germinal Pierre Dandelin（1794－1847）首先说明了一个圆柱面与一个平面之间一般相交为椭圆。考虑相交的一个圆柱面和一个平面；取一个半径与圆柱面的半径相同的球，并将它放进圆柱体内，再取一个半径相同的球，将其从圆柱体的底部放入圆柱体内。这两个球将与相交的平面相切于两个点，这两个点就是相交所得的椭圆的两个焦点，如图 11.39 所示。此外，两个球与圆柱体相切所得的两个圆之间沿圆柱面方向的最小距离刚好就是相交所得椭圆的主半径的两倍。在 Miller 和 Goldman（1992）中可以找到该结论的简略证明，其中描述了如何利用这一事实来寻找相交所得的椭圆，并用前面所说的定义方法来表示。我们在此提供了该方法。

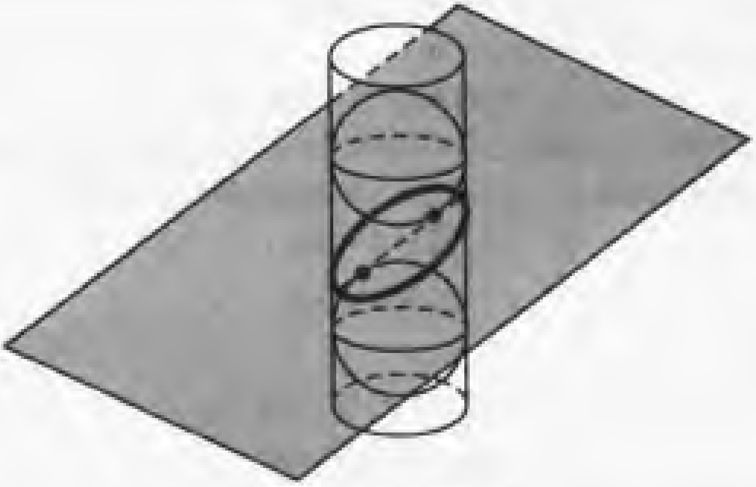


图 11.39　丹德林构图

我们假设用基点B，轴$\hat{a}$和半径r来表示圆柱面，用一个点P和法线$\hat{n}$来表示平面，因而，隐含形式的方程为

$$(X-P)\cdot\hat{n}=0 \tag{11.21}$$

图 11.40 显示了该圆柱面、与其相交的平面，以及两个球的截面图。

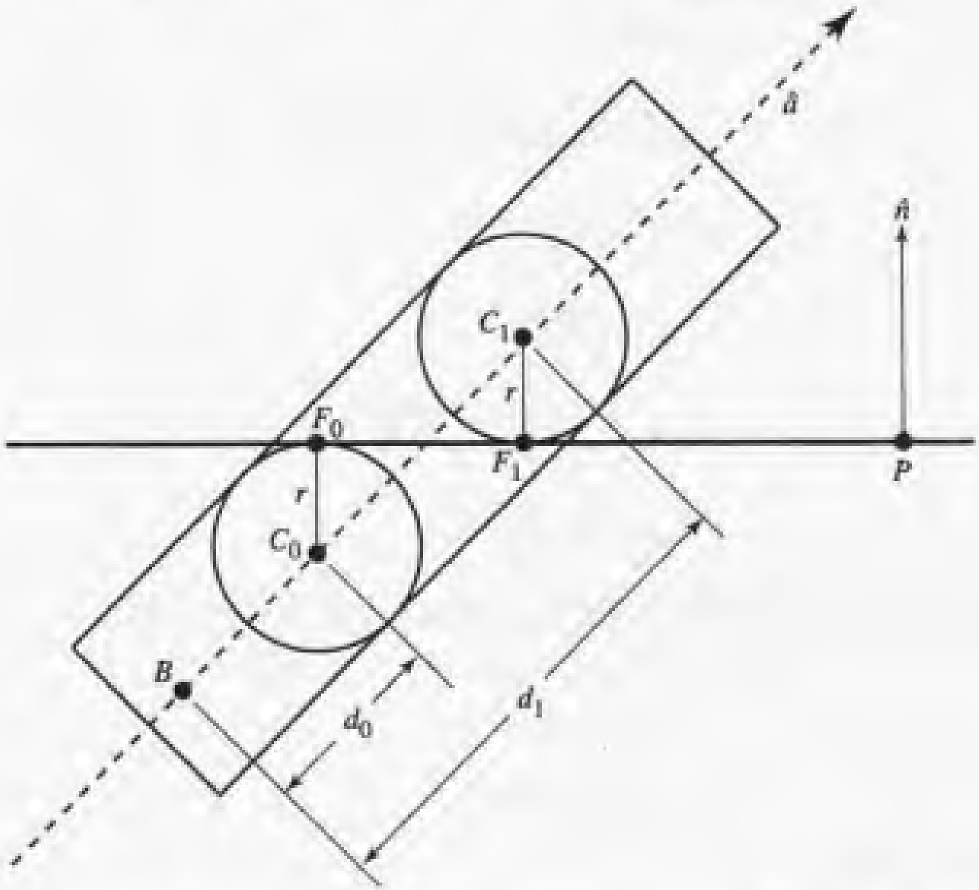


图 11.40　平面与圆柱面相交的横截面，两个球面用于定义相交的椭圆（据 Miller 和 Goldman，1992）

Miller 和 Goldman 从如下的两个事实开始证明：

- 两个球的球心必定位于圆柱面的轴上。
- 切点为$C \pm r\hat{n}$，并构成相交所得椭圆的两个焦点。

由于我们已知球的半径及其球心所在的轴，要完成它们的定义，就只需确定某个固定点（例如B点，即圆柱面的基底）与它们之间的距离d_0和d_1。我们有

$$C_0 = B + d_0\hat{a}$$

$$C_1 = B + d_1\hat{a}$$

于是，焦点为

$$\begin{aligned} F_0 &= B + d_0\hat{a} + r\hat{n} \\ F_1 &= B + d_1\hat{a} - r\hat{n} \end{aligned} \tag{11.22}$$

为了确定上述距离，我们将方程（11.22）代入方程（11.21），并求解

$$d_0 = \frac{(P - B) \cdot \hat{n} - r}{\hat{a} \cdot \hat{n}}$$

$$d_1 = \frac{(P - B) \cdot \hat{n} + r}{\hat{a} \cdot \hat{n}}$$

我们可以期望相交所得的椭圆的中心C将位于圆柱面的轴$\hat{a}$上，注意到C刚好就是两个焦点F_0 和 F_1的中点，就能理解这一点：

$$\begin{aligned} C &= \frac{F_0 + F_1}{2} \\ &= \frac{B + d_0\hat{a} + r\hat{n} + B + d_1\hat{a} - r\hat{n}}{2} \\ &= \frac{2B + (d_0 + d_1)\hat{a}}{2} \\ &= B + \frac{d_0 + d_1}{2}\hat{a} \\ &= B + \frac{\frac{(P-B)\cdot\hat{n}-r}{\hat{a}\cdot\hat{n}} + \frac{(P-B)\cdot\hat{n}+r}{\hat{a}\cdot\hat{n}}}{2}\hat{a} \\ &= B + \frac{(P - B) \cdot \hat{n}}{\hat{a} \cdot \hat{n}}\hat{a} \end{aligned}$$

点C是上述平面与圆柱面的轴$\hat{a}$的交点。主轴的方向$\hat{u}$平行于$F_0 - F_1$：

$$\begin{aligned} F_0 - F_1 &= B + d_0\hat{a} + r\hat{n} - (B + d_1\hat{a} - r\hat{n}) \\ &= (d_0 + d_1)\hat{a} + 2r\hat{n} \\ &= \frac{-2r}{\hat{a} \cdot \hat{n}}\hat{a} + 2r\hat{n} \end{aligned}$$

改写该式，可得

$$\frac{\hat{a} \cdot \hat{n}}{-2r}(F_0 - F_1) = \hat{a} - (\hat{a} \cdot \hat{n})\hat{n}$$

Miller 和 Goldman 从上式中发现，主轴的方向就是$\hat{a}$的垂直于$\hat{n}$的分量。如前所述，主轴就是球与圆柱面相切的两个圆之间的距离（即$|d_1 - d_0|$）的一半，如图 11.40 所示，于是我们得到

$$r_u = \frac{|d_1 - d_0|}{2} = \frac{r}{|\hat{a} \cdot \hat{n}|}$$

次半径r_v等于圆柱面的半径r。为了证明该点，我们从如下的事实出发：如果我们有两个椭圆：

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

其中$a > b$，可以看出，两个焦点为$\pm\sqrt{a^2 - b^2}$。因此，如果我们定义c为中心C与焦点之间的距离，那么此半径为

$$r_v = \sqrt{r_u^2 - c^2}$$

两个焦点之间的距离为$2c$，因此距离平方为

$$4c^2 = |F_0 - F_1|^2 = 4r^2\left(\frac{1}{(\hat{a} \cdot \hat{n})^2} - 1\right)$$

两边取平方根，并代入$\cos(\theta) = \hat{a} \cdot \hat{v}$，可得

$$2c = |F_0 - F_1| = \sqrt{4r^2\left(\frac{1}{\cos^2(\theta)} - 1\right)} = 2r\tan(\theta)$$

因此，我们有

$$r_v = \sqrt{r_u^2 - c^2} = \sqrt{\left(\frac{r}{|\hat{a} \cdot \hat{n}|}\right)^2 - \left(\frac{2r\tan(\theta)}{2}\right)^2} = \sqrt{\left(\frac{r}{\cos(\theta)}\right)^2 - (r\tan(\theta))^2} = \sqrt{r^2\frac{1 - \sin^2(\theta)}{\cos^2(\theta)}} = r$$

前面提到过，平面与圆柱面相交的特殊情形可能是一个圆或者一条或两条直线，我们在此会进行介绍。很清楚，如果平面的法线$\hat{n}$与圆柱面的轴$\hat{a}$平行，那么相交为一个圆，如图 11.41 所示。

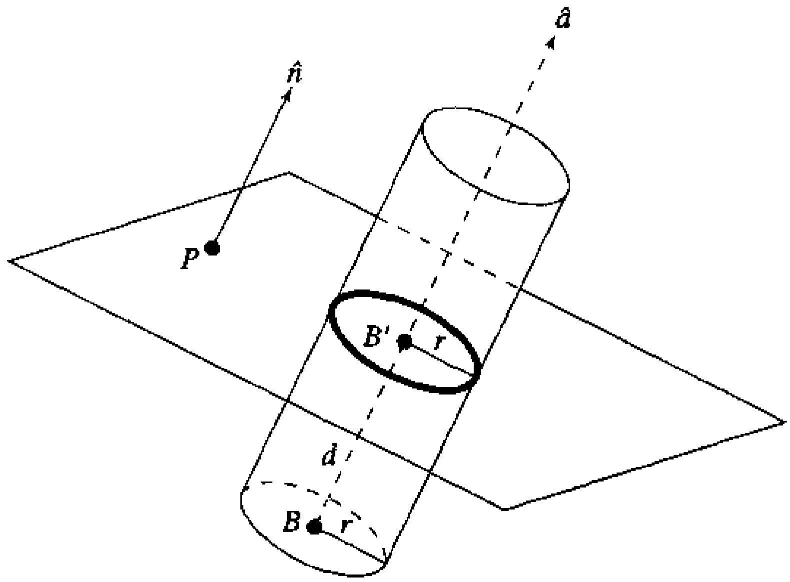


图 11.41　如果平面的法线与圆柱的轴平行，那么平面与圆柱相交为一个圆

如果平面的法线$\hat{n}$与圆柱面的轴$\hat{a}$垂直，那么我们计算圆柱面的基点B与平面之间的距离：

- 如果$|d| > r$，那么它们不相交。
- 如果$|d| = r$，那么它们相交为一条直线$B' + t\hat{a}$，其中B'是B在平面上的投影。
- 如果$|d| < r$，那么它们相交的图形包含两条直线：

$$B' \pm \sqrt{r^2 - d^2}(\hat{a} \times \hat{n}) + t\hat{a}$$

伪码为

```
boolean PlaneCylinderIntersection(Plane plane, Cylinder cylinder)
{
    // Compute distance, projection of cylinder base point onto
    // plane, and angle between plane normal and cylinder axis
    d = PointPlaneDistance(cylinder.base, plane);
    bPrime = cylinder.base - d * plane.normal;
    cosTheta = Dot(cylinder.axis, plane.normal);

    // Check angle between plane and cylinder axis to
    // determine type of intersection

    if (cosTheta < abs(epsilon)) {
        // No intersection, or one or two lines. Check
        // which it is by looking at distance from cylinder
        // base point to plane
        if (abs(d) == cylinder.radius) {
            // Single line
            line.base = bPrime;
            line.direction = cylinder.axis;
            return true;
        }
```

```
        if (abs(d) > cylinder.radius) {
            // No intersection
            return false;
        }

        // abs(d) < cylinder.radius, so two intersection lines
        offset = Cross(cylinder.axis, plane.normal);
        e = sqrt(cylinder.radius * cylinder.radius - d * d);

        Line line1, line2;

        line1.base = bPrime - e * offset;
        line1.direction = cylinder.axis;
        line2.base = bPrime + e * offset;
        line2.direction = cylinder.axis;

        return true;
    }

    // cosTheta != 0, so intersection is circle or ellipse
    if (abs(cosTheta) == 1) {
        // Circle
        Circle circle;
        circle.center = bPrime;
        circle.normal = cylinder.axis;
        circle.radius = cylinder.radius;

        return true;
    }

    // abs(cosTheta) != 0 and abs(cosTheta) != 1, so ellipse
    Ellipse ellipse;
    ellipse.center = bPrime - (d / cosTheta) * cylinder.axis;
    ellipse.u = cylinder.axis - cosTheta * plane.normal;
    ellipse.v = Cross(plane.normal, ellipse.u);
    rU = cylinder.radius / abs(cosTheta);
    rV = cylinder.radius;

    return true;
}
```

11.7.4　平面与圆锥面的相交

在本节中，我们讨论计算平面与圆锥面相交的问题，如图 11.42 所示。实际上，圆锥面与平面之间的相交方式有许多种，图 11.43 中显示了其中的 8 种。注意这些相交都是相对于有限的圆锥面的。无限的圆锥面具有较少的相交构形，因为我们不需要处理端面。我们给出了平面与无限圆锥面及平面与有限圆锥面之间的相交的检测算法。前者更简单些。在随后的一个子节中，我们将给出一种计算平面与无限圆锥面相交的算法。通过用包

含端面的平面来“裁剪”相交的二次曲线或直线的范围，可以将这类算法扩展，用来处理有限圆锥面。

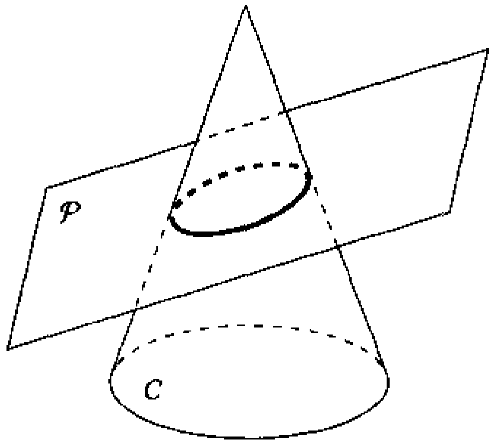


图 11.42　平面与圆锥面的相交

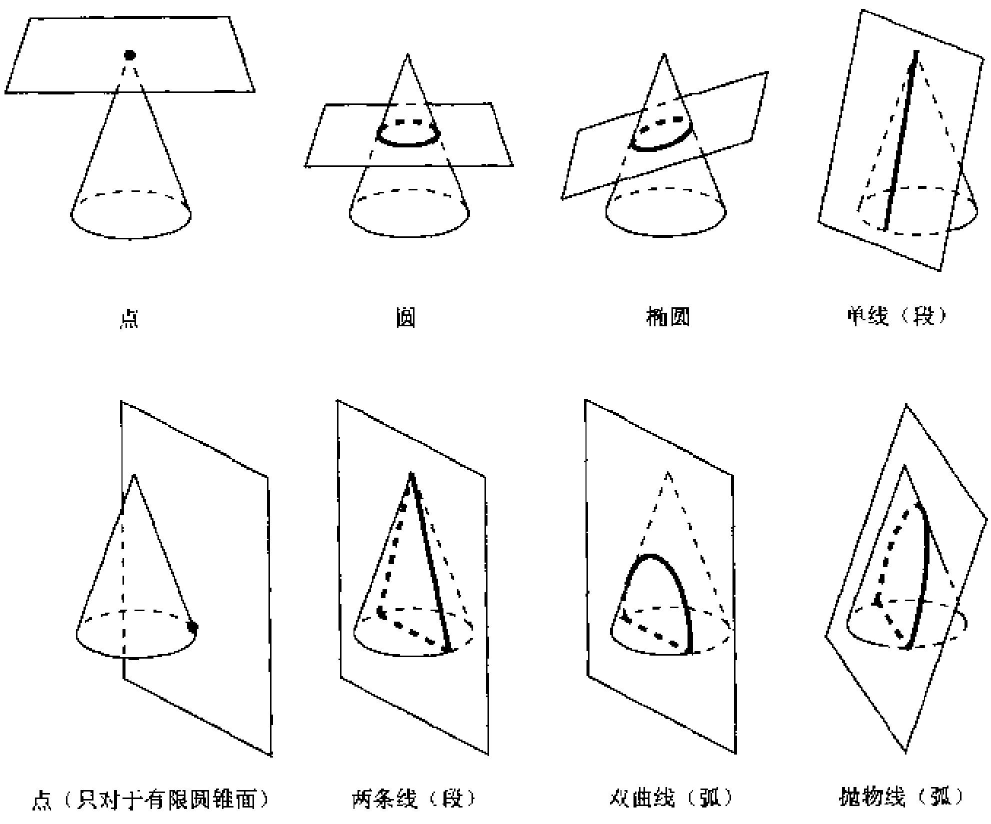


图 11.43　平面与圆锥面相交的几种方式

1. 相交检测

为了描述相交检测算法，我们用隐含形式来定义平面：

$$\boldsymbol{P}\cdot\hat{\boldsymbol{n}}+d=0$$

（用坐标来表示的形式一般为$ax+by+cz+d=0$，其中$\sqrt{a^2+b^2+c^2}=1$）。圆锥面

定义于“一般位置”——即用基点B，轴$\hat{d}$和高h来表示（参见 11.3.5 节中的图 11.14）。一个无限的单圆锥可用定义圆锥的顶点的点A，一个轴$\hat{d}$和半高α来定义（如图 11.44 所示）。

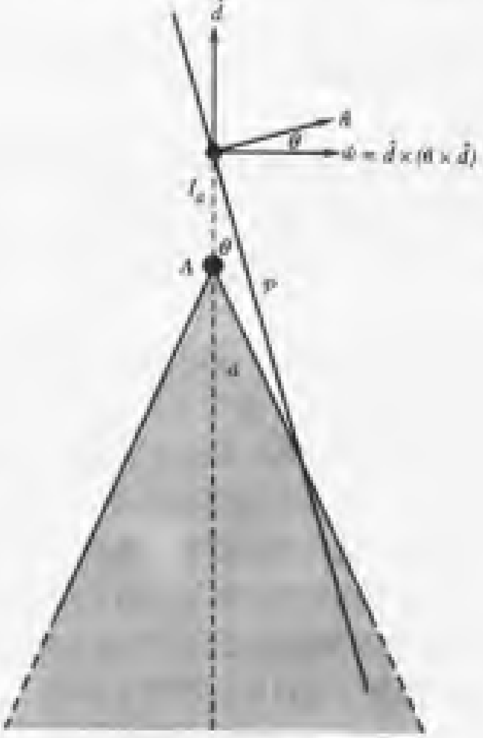


图 11.44　平面与无限圆锥面的相交检测

（1）无限圆锥面

基于检测相交的目的，平面$\mathcal{P}$和圆锥面C可以是如下几种构形中的一种。

① 如果$\mathcal{P}$平行于C的轴，$\hat{d} \cdot \hat{n} = 0$，则它们总是相交的。

② 如果$\mathcal{P}$垂直于C的轴，$|\hat{d} \cdot \hat{n}| = 1$，那么如果从圆锥面的顶点到平面的（有符号）距离是非负值（相对于$\hat{d}$），则它们相交。

③ 如果$\mathcal{P}$与C的轴既不垂直也不平行，那么它们不一定相交，是否相交取决于从A到圆锥面的轴与平面的交点之间的距离以及圆锥面的轴与平面之间的夹角。

图 11.44 显示了一个圆锥面与一个平面的相交检测，图中显示的是平面与圆锥面的轴既不垂直也不平行的情形。垂直于$\mathcal{P}$且通过A的平面$\mathcal{P}^{\perp}$平行于$\hat{d}$，因此其法线为

$$\hat{n} \times \hat{d}$$

我们定义$\mathcal{P}^{\perp}$上垂直于$\hat{d}$的一个向量为

$$\hat{w} = \hat{d} \times (\hat{n} \times \hat{d})$$

$\hat{n}$与$\hat{w}$之间的夹角θ为

$$\cos(\theta) = \hat{n} \cdot \vec{w}$$

如果平面与圆锥面的轴的交点位于圆锥面内部（上述有符号距离小于或等于零），那么显然它们相交。否则，如果$\theta \leqslant \alpha$，则它们相交。

（2）有限圆锥面

基于检测相交的目的，平面$\mathcal{P}$和圆锥面C可以是如下几种构形中的一种。

① $\mathcal{P}$可能平行于C的轴：$|\hat{d}\cdot\hat{n}|=1$。

② $\mathcal{P}$可能垂直于C的轴：$\hat{d}\cdot\hat{n}=0$。

③ $\mathcal{P}$可能与C的轴既不平行也不垂直。在这种情形中，$\mathcal{P}$与C不一定相交。

我们分别考虑这些情形。

① 如果$\mathcal{P}$平行于C的轴，那么，如果$\mathcal{P}$与C的轴之间的距离小于或等于圆锥面的半径（此时的相交为一条二次曲线或一条直线），则它们相交。

② 如果$\mathcal{P}$垂直于C的轴，那么，如果$\mathcal{P}$与B之间的距离（相对于$\hat{d}$）位于0和h之间，则它们相交。

③ 如果$\mathcal{P}$与C的轴既不平行也不垂直，那么需要考虑两种情形：

a．$\mathcal{P}$与B之间的距离（相对于$\hat{d}$）位于0和h之间，此时它们一定相交。

b．$\mathcal{P}$与C的轴的相交位于圆锥面的顶点和端面外，此时不一定相交，是否相交取决于交点的相对位置，以及平面与轴之间的夹角。

除最后一种情形外，其他所有情形都很简单。确定平面与圆锥面的轴是否平行或垂直仅用一次点积就能解决。计算圆锥面的基点B与平面之间的距离也很简单并且不费时（参见10.3.1节）。计算平面与圆锥面的轴的相交也不费时（参见11.1.1节）。只有最后一种情形涉及费时的操作，因此必须根据上面讨论的次序来解决。

最后一种情形显示如图11.45所示。显示为横截图并不仅仅是为了图示的方便——确定它们是否相交的方法是在一个通过B且与$\mathcal{P}$垂直的平面上进行的。其余的都是简单的三角几何。

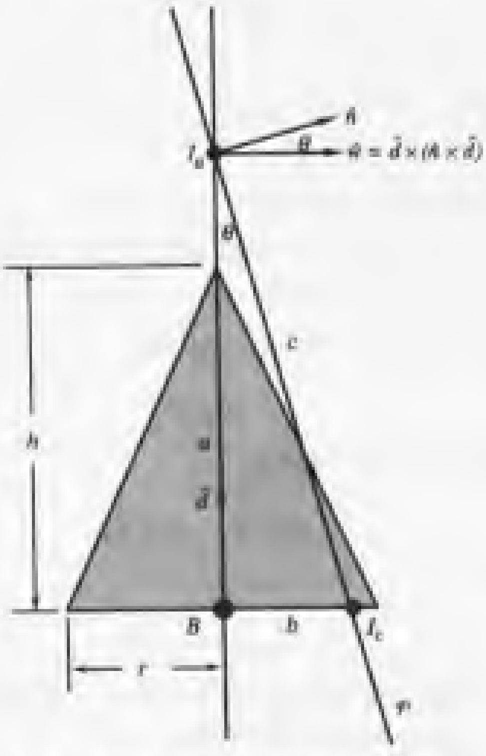


图11.45　平面—圆锥面相交的边缘视图

如果I_a是$\mathcal{P}$与圆锥面的轴的相交图形，那么如果点I_c与轴之间的距离小于r（圆锥面的半径），则存在相交。垂直于$\mathcal{P}$且通过B的平面$\mathcal{P}^\perp$平行于$\hat{d}$，因此其法线为

$$\hat{n} \times \hat{d}$$

我们定义$\mathcal{P}^\perp$上垂直于$\hat{d}$的一个向量为

$$\hat{w} = \hat{d} \times (\hat{n} \times \hat{d})$$

$\hat{n}$与$\hat{w}$之间的夹角θ为

$$\cos(\theta) = \hat{n} \cdot \vec{w}$$

我们还知道距离a为：

$$a = \|I_a - B\| - h$$

根据余弦函数的定义，我们知道

$$\cos(\theta) = \frac{a}{c}$$

代入可得

$$\hat{n} \cdot \vec{w} = \frac{\|I_a - B\| - h}{c}$$

因此有

$$c = \frac{\|I_a - B\| - h}{\hat{n} \cdot \vec{w}}$$

调用毕达哥拉斯定理，可得

$$a^2 + b^2 = c^2$$

$$(\|I_a - B\| - h)^2 + b^2 = \left(\frac{\|I_a - B\| - h}{\hat{n} \cdot \vec{w}}\right)^2$$

$$b^2 = \left(\frac{\|I_a - B\| - h}{\hat{n} \cdot \vec{w}}\right)^2 - (\|I_a - B\| - h)^2$$

因此，如果$b^2 \leqslant r^2$，则存在相交；否则不存在相交。

2. 与无限圆锥面的相交

在本节中，我们将讨论平面与无限圆锥面之间的相交问题。在此，我们用平面上的一个点P和平面的法线$\hat{n}$来定义平面；用圆锥的顶点V，轴$\hat{a}$和半角α来定义圆锥面，如图 11.46 所示。

与处理平面与圆柱面的相交问题（参见 11.7.3 节）一样，我们提供 Miller 和 Goldman（1992）提出的一种方法。在这种方法中，他们利用了丹德林构图——非退化的相交是与在内部与圆柱面相切且与平面相切的球面具有特殊关系的一条曲线。对于一个平面与一个无限圆锥面的相交来说，可以利用一种相似的技术。我们在前面已指出（见图 11.43），平面与圆锥面可能相交于一个点、一条直线、两条直线、一个圆、一个椭圆、一条抛物线或一条双曲线。前 3 种情形都是所谓的“退化情形”，计算这类相交非常简单。后面的 4 种情形是一般情形，我们使用类似于计算平面－圆柱面相交的方法。

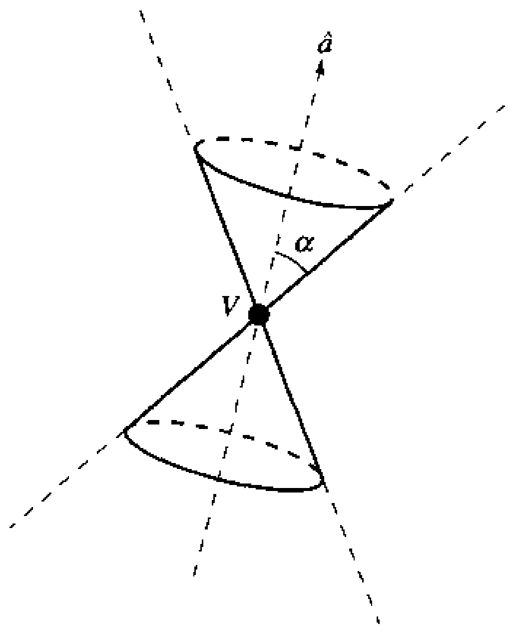


图 11.46　无限圆锥面的定义

在讨论一个平面与一个圆柱面的相交问题时，图 11.37 和图 11.38 分别显示了椭圆和圆的定义。对于抛物线和双曲线，我们可得到相似的几何定义：抛物线用其顶点V_P，准线和焦点向量$\hat{u}$ 和$\hat{v}$，以及焦距f（顶点与焦距之间的距离）来定义；双曲线用其中心点C，主轴$\hat{u}$ 和次轴$\hat{v}$，以及相关的主半径r_u和次半径r_v来定义：

$$r_u \text{和} r_v = \sqrt{d^2 - r_u^2}$$

其中d是中心点C与焦点F_1和F_2之间的距离，如图 11.47 所示。

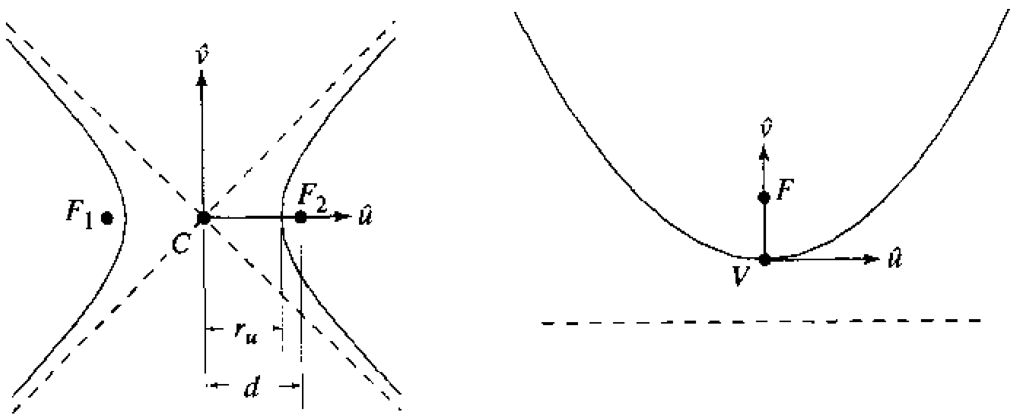


图 11.47　双曲线和抛物线的几何定义

对于非退化的相交来说，可以找到与圆锥面相切于一个圆（即它们位于圆锥面内）并且与相交的平面相切的一个或两个球。如果这个平面与该圆锥面相交于一条抛物线，那么存在一个这样的球，并且这个球与该平面接触于抛物线的焦点。如果这个平面与该圆锥面相交于一条双曲线或一个椭圆，那么存在两个这样的球，并且这个球与该平面相切于相交所得曲线的焦点。与平面一圆柱面相交的情形一样，两个球与圆锥面相切所得的两个圆之间沿圆锥面方向的距离刚好就是相交所得椭圆的主半径的两倍。在 Miller 和 Goldman

（1992）中可以找到上述结论的简略证明。

可以很简单地用一个共有的性质来说明导致退化相交（点、一条直线、两条直线）的构形——当圆锥面的顶点与平面相交时，将得到退化相交。考虑图 11.43：我们很容易看出产生“点”、“一条直线（线段）”和“两条直线（线段）”的情形。应该注意，在实现代码中，圆锥面的顶点位于平面上的检测不应该是绝对的——应该利用一定的容差ϵ；否则，将得到至少有一个参数（例如主半径）非常小的一条二次曲线，这并不是我们期望的计算结果。

（1）平面 - 圆锥面的非退化相交

为了区分不同的相交，我们考虑圆锥面的轴$\hat{a}$和平面的法线$\hat{n}$之间的夹角，以及它与定义圆锥面的半角α之间的关系。我们定义圆锥面的轴$\hat{a}$和平面的法线$\hat{n}$之间的夹角为θ。根据点积的定义，我们有$\cos(\theta) = \hat{a}$。

为了利用相切的球的性质，我们首先需要确定，相对于圆锥面和平面的方向，相切的球位于何处。通过确定球与圆锥面相切的条件，以及球与平面相切的条件，并将其中一个方程代入另一个方程，可以得到球同时与圆锥面和平面相切的条件。为了简化这种情形，做如下假设：

$$(V - P)\cdot\hat{n} < 0$$
$$\hat{a}\cdot\hat{n} \geqslant 0$$

如果不能满足其中的任一个假设，只需反转$\hat{n}$和/或$\hat{a}$。

观察图 11.48，很清楚，球$\{C, r\}$位于直线$V + t\hat{a}$上；利用正弦函数的定义，我们有$h = \frac{r}{\sin(\alpha)}$，因而可得

$$C = V + \frac{r}{\sin(\alpha)}\hat{a} \tag{11.23}$$

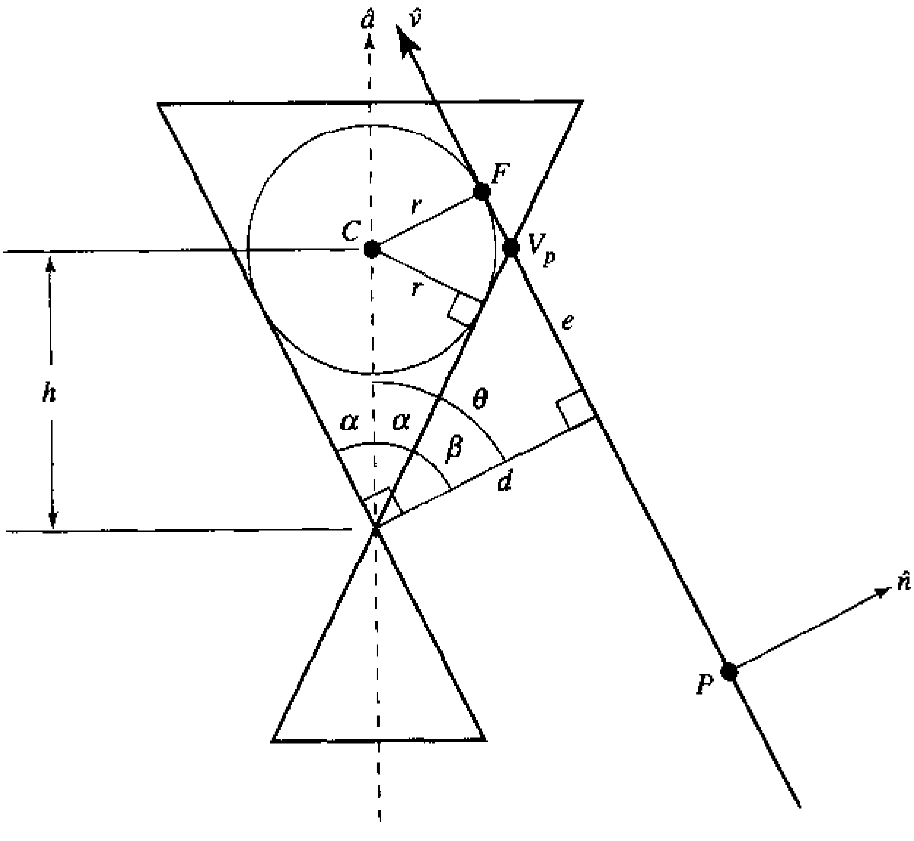


图 11.48　平面与圆锥面的抛物线相交（Miller 和 Goldman，1992）

Miller 和 Goldman 注意到，如果允许 r 为负数，那么得到的圆将位于圆锥面的另一边。

对于球与平面相切的情形，根据定义，我们知道球的球心 C 与平面的距离一定为 r，即

$$\|(C-P)\cdot\hat{n}\| = |r|$$

或者，两边同时平方，可得

$$((C-P)\cdot\hat{n})^2 = r^2 \tag{11.24}$$

如果将方程（11.23）代入方程（11.24），我们得到

$$\left(\left((V-P)+\frac{r}{\sin(\alpha)}\hat{a}\right)\cdot\hat{n}\right)^2 = r^2$$

这是一个关于 r 的二次方程，其解为

$$r = \pm\frac{((V-P)\cdot\hat{n})\sin(\alpha)}{\sin(\alpha)\mp\cos(\theta)} \tag{11.25}$$

如果平面平行于圆锥面的水线，则它们之间的相交为一条抛物线，即当 $\theta+\alpha=\pi/2$ 时，如图 11.48 所示，相交为抛物线。注意，在方程（11.25）中，r 只可能有一个解，因为当 $\theta+\alpha=\pi/2$ 时，有 $\cos(\theta)=\sin(\alpha)$，而且方程（11.25）的分母中有一个为 0。如果 $\cos(\theta)=\sin(\alpha)$，则方程（11.25）变为

$$\begin{aligned} r &= -\frac{(V-P)\cdot\hat{n}\sin(\alpha)}{\sin(\alpha)+\cos(\theta)} \\ &= -\frac{(V-P)\cdot\hat{n}\sin(\alpha)}{2\sin(\alpha)} \\ &= -\frac{(V-P)\cdot\hat{n}}{2} \end{aligned}$$

圆锥面的顶点与平面之间的距离为

$$d = -(V-P)\cdot\hat{n}$$

因而，可得

$$r = \frac{d}{2} \tag{11.26}$$

为了定义该抛物线，我们需要得到焦距长 f，顶点 V_P，以及基底 $\hat{u}$ 和 $\hat{v}$。我们通过确定 F（抛物线的焦点）并计算它与顶点 V_P 之间的距离来确定 f：

$$f = \|F - V_p\|$$

点 V_P 和 F 位于包含 V，$\hat{a}$ 和 $\hat{n}$，以及 $F - V_p \parallel \hat{v}$ 的平面上。因此，$\hat{v}$ 也位于该平面上。由于它还必定垂直于该平面的法线 $\hat{n}$，因此我们有

$$\begin{aligned} \hat{v} &= \frac{\hat{a}-(\hat{a}\cdot\hat{n})\hat{n}}{\|\hat{a}-(\hat{a}\cdot\hat{n})\hat{n}\|} \\ &= \frac{\hat{a}-\cos(\theta)\hat{n}}{\|\hat{a}-\cos(\theta)\hat{n}\|} \end{aligned}$$

其中给出了 V_P 和 F 所确定的方向。为了确定这些点的位置，我们需要计算值 e。注意到，

$\beta = \theta - \alpha = \pi/2 - 2\alpha$ 且$\tan(\beta) = e/a$。将它们组合在一起，可得

$$\begin{aligned} e &= d\tan(\beta) \\ &= d\tan(\frac{\pi}{2} - 2\alpha) \\ &= d\cot(2\alpha) \\ &= d\cot(\pi - 2\theta) \\ &= -d\cot(2\theta) \\ &= d\left(\frac{\tan(\theta) - \cot(\theta)}{2}\right) \end{aligned}$$

通过观察图 11.48，我们看到平面与圆锥面的轴的交点与圆锥面的顶点之间的距离为 $2h$，因此，我们有

$$\cos(\theta) = \frac{2h}{d}$$

或者

$$h = \frac{d}{2\cos(\theta)}$$

至此，我们可以计算出抛物线的顶点和焦点：

$$V_p = V + d\hat{n} + e\hat{v}$$

$$F = V + h\hat{a} + r\hat{v}$$

利用这些公式，我们可以很容易地计算出焦距f。然而，我们可以得到更简洁的公式。考虑三角形ΔFCV_p：这是一个直角三角形，其锐角为$\gamma = \pi/2 - \theta$，显然它的一条直角边为r。用x来表示另一条直角边。根据三角几何，我们有

$$\tan(\gamma) = \frac{x}{r}$$

然而，我们知道$r = d/2$（方程 11.26），因此，上式可以改写为

$$\tan(\pi/2 - \theta) = \frac{x}{\frac{d}{2}}$$

回顾三角几何中的如下公式：

$$\cos(\frac{\pi}{2} - \alpha) = \sin(\alpha)$$

$$\sin(\frac{\pi}{2} - \alpha) = \cos(\alpha)$$

$$\tan(\alpha) = \frac{\sin(\alpha)}{\cos(\alpha)}$$

我们可以将它们改写为

$$\frac{1}{\tan(\theta)} = \frac{x}{\frac{d}{2}}$$

或者

$$x = \frac{d}{2}\cot(\theta)$$

处理这种情形的伪码如下（Miller 和 Goldman 1992）：

```
float d = Dot(plane.base - cone.vertex, plane.normal);
float cosTheta = Dot(plane.normal, cone.axis);
float sinTheta = Sqrt(1 - cosTheta * cosTheta);
float tanTheta = sinTheta / cosTheta;
float cotTheta = 1 / tanTheta;
float e = d/2 * (tanTheta - cotTheta);

// Parabola is {V, u, v, f}

Vector v = Normalize(cone.axis - cosTheta * plane.normal);
Vector u = Cross(v, plane.normal);
Point  V = cone.vertex + d * plane.normal + e * v;
float  f = d/2 * cotTheta;
```

如果平面的法线$\hat{n}$平行于圆锥面的轴$\hat{a}$，则相交为一条圆形曲线，即如果$|\hat{n}\cdot\hat{a}|<\epsilon$时，如图 11.49 所示，相交为圆形曲线。可以非常简单地计算出圆：显然，圆的圆心为

$$C = V - h\hat{a}$$

其中h是平面与圆锥面的顶点V之间的（有符号）距离。圆的法线当然就是平面的法线$\hat{n}$。可用如下的简单三角公式来计算半径r：

$$r = \|h\|\tan(\alpha)$$

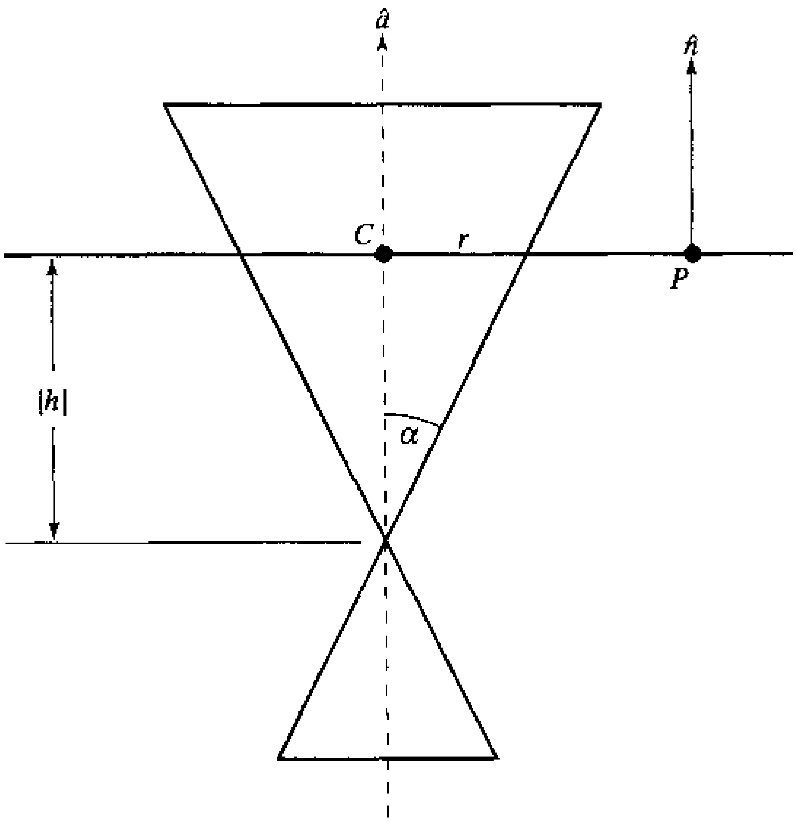


图 11.49　平面与圆锥面的圆形相交（Miller 和 Goldman 1992）

处理这种情形的伪码如下（Miller 和 Goldman 1992）：

```
// Signed distance from cone's vertex to the plane
h = Dot(cone.vertex - plane.point, plane.normal);

circle.center = cone.vertex - h * plane.normal;
circle.normal = plane.normal;
circle.radius = Abs(h) * Tan(cone.alpha);
```

当$\cos(\theta) \neq \sin(\alpha)$时，得到的是椭圆形相交，此时平面的法线与圆锥面的轴不平行，如图 11.50 和图 11.51 所示。如果$\cos(\theta) > \sin(\alpha)$，则相交为一个椭圆；如果$\cos(\theta) < \sin(\alpha)$，则相交为一条双曲线。为了定义椭圆或双曲线，我们首先必须确定它们的中心C。根据定义，存在一个位于两个焦点之间的中点：$C = (F_0 + F_1)/2$。

即有

$$F_0 = V + \frac{r_0}{\sin(\alpha)} - r_0\hat{n}$$

$$F_1 = V + \frac{r_1}{\sin(\alpha)} - r_1\hat{n}$$

其中r_0和r_1为方程（11.25）中的两个半径。在椭圆的情形中，根据$\cos(\theta)$和$\sin(\alpha)$之间的关系，可以得到正的r_0和r_1，并且球位于平面的另一面，但总是位于圆锥面的同一面；在双曲线的情形中，根据$\cos(\theta)$和$\sin(\alpha)$之间的关系，可以得到负的r_0和正的r_1，并且球位于平面的同一面，但总是位于圆锥面的另一面。

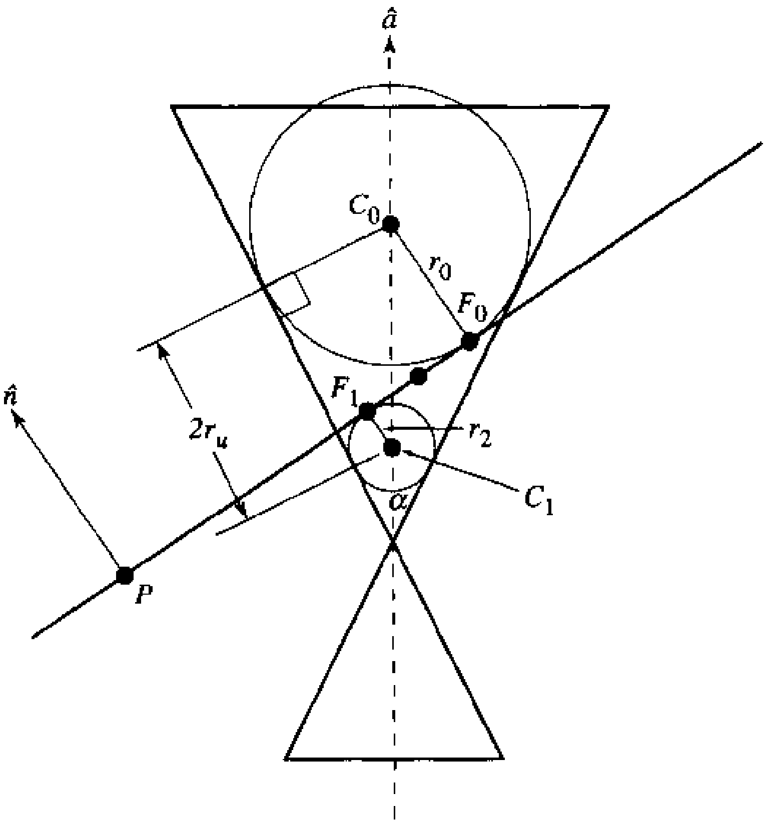


图 11.50　平面与圆锥面的椭圆形相交（Miller 和 Goldman 1992）

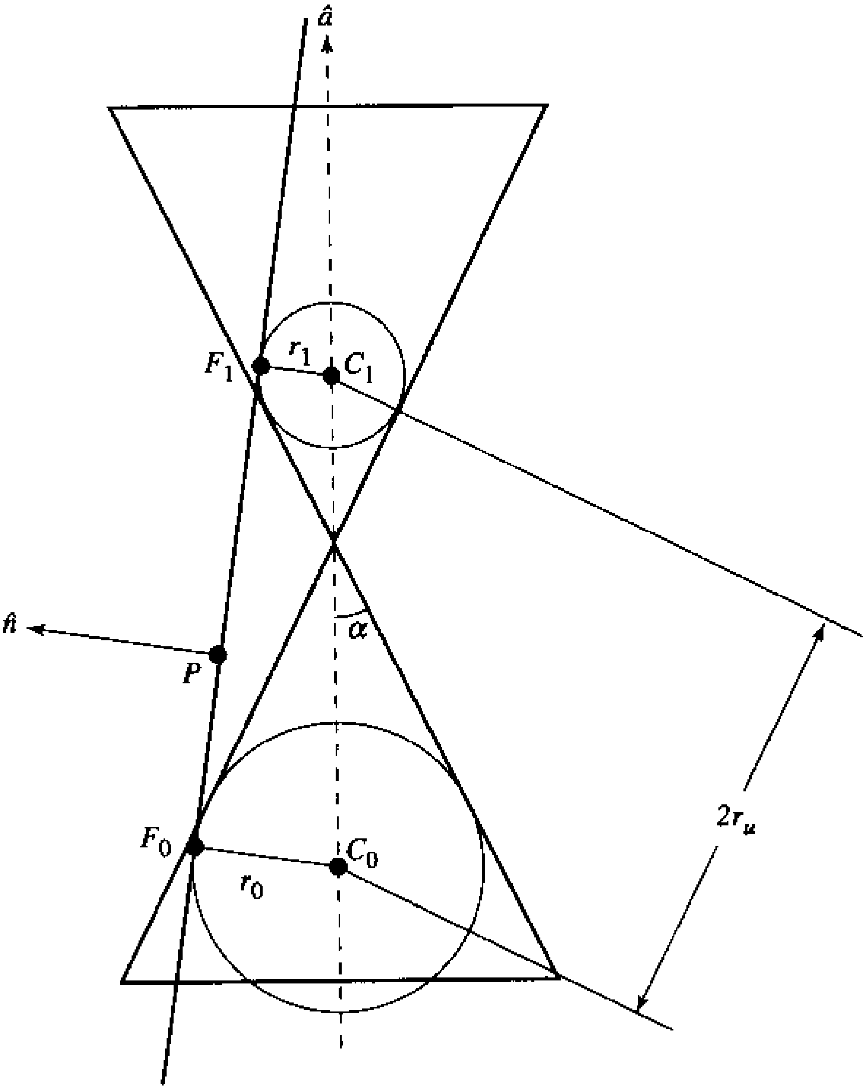


图 11.51　平面与圆锥面的双曲线形相交（Miller 和 Goldman 1992）

如果我们将方程（11.25）的上述两个版本代入计算焦距的公式中，然后再将得到的结果代入计算中心的公式中，经过一些运算，可得

$$C = V + h\cos(\theta)\hat{a} - h\sin^2(\alpha)\hat{n}$$

其中

$$t = (P - V)\cdot\hat{n}$$

$$b = \cos^2(\theta) - \sin^2(\alpha)$$

$$h = \frac{t}{b}$$

下面计算准线和焦点向量$\hat{u}$和$\hat{v}$。如果我们注意到C，F_0和F_1都位于与包含$\hat{a}$和$\hat{n}$的平面族平行的平面上，那么我们可以看出

$$\hat{u} = \frac{\hat{a} - \cos(\theta)\hat{n}}{\|\hat{a} - \cos(\theta)\hat{n}\|}$$

因此有

$$\hat{v} = \hat{n} \cdot \hat{u}$$

主半径和次半径的计算更复杂一些，我们完全采用 Miller 和 Goldman（1992）的方法：前面已经提到，用于圆锥面一平面相交问题的两个相切的球具有如下的性质，即主半径为两个球与圆锥面相切所得的两个圆之间沿着水线方向的距离的一半。对于椭圆的情形，我们可以得到

$$\begin{aligned} r_u &= \frac{1}{2}\left(\frac{r_0}{\tan(\alpha)} - \frac{r_1}{\tan(\alpha)}\right) \\ &= h\sin(\alpha)\cos(\alpha) \end{aligned}$$

对于双曲线的情形，我们有

$$\begin{aligned} r_u &= \frac{1}{2}\left(\frac{r_1}{\tan(\alpha)} - \frac{r_0}{\tan(\alpha)}\right) \\ &= h\sin(\alpha)\cos(\alpha) \end{aligned}$$

注意，在上述两种情形中，r_u 都为正。

如果我们用 d 表示中心C与焦点之间的（正的）距离，那么椭圆的次半径为

$$r_v = \sqrt{r_u^2 - d_u^2}$$

对于双曲线的情形，我们有

$$r_v = \sqrt{d_u^2 - r_u^2}$$

注意，在上述两种情形中，r_v 都为正。

中心C位于两个焦点的中点处，因此，d 是两个焦点之间距离的一半，因此有

$$d^2 = \frac{1}{4}(F_0 - F_2) \cdot (F_0 - F_2)$$

如果我们将计算F_0和F_1的公式代入，并将计算$r_0 + r_1$和$r_0 - r_1$的公式代入，可得

$$r_v = t\frac{\sin(\alpha)}{\sqrt{b}}$$

伪码如下所示（Miller 和 Goldman 1992）：

```
// Compute various angles
cosTheta = Dot(cone.axis, plane.normal);
cosThetaSqared = cosTheta * cosTheta;
sinAlpha = Sin(cone.alpha);
sinAlphaSquared = sinAlpha * sinAlpha;
cosAlpha = Sqrt(1 - sinAlphaSquared);

t = Dot(plane.point - cone.vertex, plane.normal);
b = cosThetaSquared - sinAlphaSquared;
h = t/b;

// Output is ellipse or hyperbola
```

```
center = cone.vertex + h * cosTheta * cone.axis
                     - h * sinAlphaSquared * plane.normal;
majorAxis = Normalize(cone.axis - cosTheta * plane.normal);
minorAxis = Cross(plane.normal, ellipse.u);
majorRadius = Abs(h) * sinAlpha * cosAlpha;
minorRadius = t * sinAlpha / Sqrt(Abs(b));
if (cosTheta > sinAlpha) {
    // Ellipse
    ellipse.center = center;
    ellipse.majorAxis = majorAxis;
    ellipse.minorAxis = minorAxis;
    ellipse.majorRadius = majorRadius;
    ellipse.minorRadius = minorRadius;

    return ellipse;
} else {
    // Hyperbola
    hyperbola.center = center;
    hyperbola.majorAxis = majorAxis;
    hyperbola.minorAxis = minorAxis;
    hyperbola.majorRadius = majorRadius;
    hyperbola.minorRadius = minorRadius;

    return hyperbola;
}
```

（2）平面－圆锥面的退化相交

为了区分3种不同的退化相交，我们考虑圆锥面的轴$\hat{a}$和平面的法线$\hat{n}$之间的夹角，以及它与定义圆锥面的半角α之间的关系。在每一种情形中，平面都包含圆锥面的顶点。退化相交如图11.52所示。

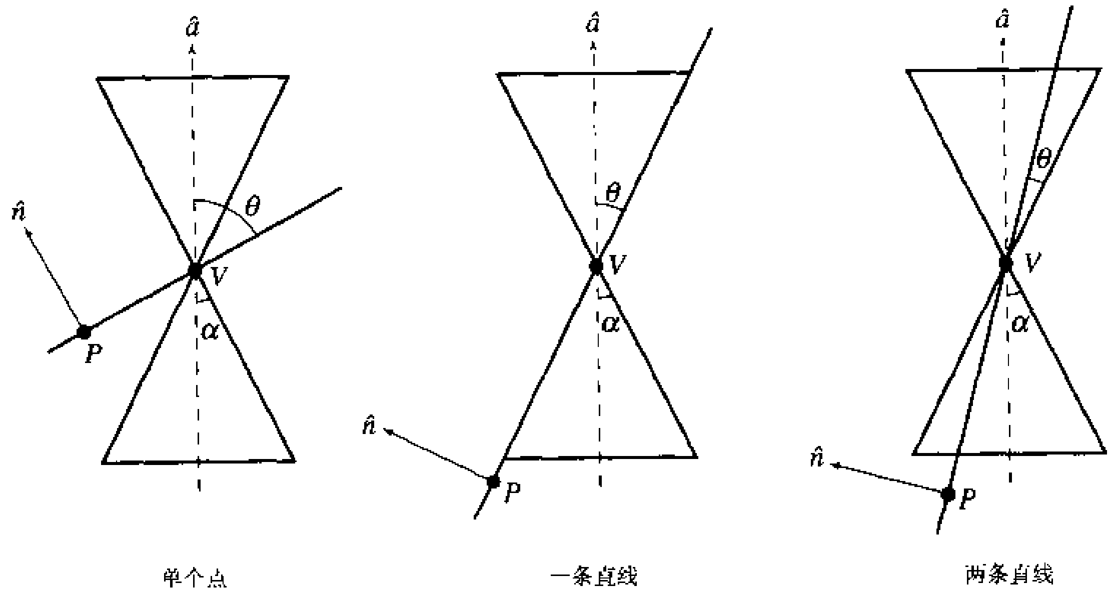


图11.52　平面与圆锥面的退化相交（Miller 和 Goldman 1992）

如果$\cos(\theta) > \sin(\alpha)$，则相交为一个点。

如果$\cos(\theta) = \sin(\alpha)$，则相交为一条直线。显然，这条直线经过圆锥面的顶点V，因此该顶点可以作为相交所得直线的基点。由于该直线位于与包含$\hat{a}$和$\hat{v}$的平面族平行的平面上，因此其方向为

$$\frac{\hat{a} - \cos(\theta)\hat{n}}{\|\hat{a} - \cos(\theta)\hat{n}\|}$$

如果$\cos(\theta) < \sin(\alpha)$，则相交图形是两条直线。这两条直线也经过圆锥面的顶点V，因此该顶点可以作为相交所得直线的基点。正如 Miller 和 Goldman 所指出的，这两条直线可以看成是一条退化的双曲线的渐近线：

$$\hat{d} = \frac{\hat{u} \pm \frac{r_v}{r_u}\hat{v}}{\|\hat{u} \pm \frac{r_v}{r_u}\hat{v}\|}$$

根据上一节的内容，我们有

$$r_u = \frac{((P - V) \cdot \hat{n}) \sin(\alpha) \cos(\theta)}{\sin^2(\alpha) - \cos^2(\theta)}$$

$$r_v = \frac{((P - V) \cdot \hat{n}) \sin(\alpha)}{\sqrt{\sin^2(\alpha) - \cos^2(\theta)}}$$

因而，可得

$$\frac{r_v}{r_u} = \frac{\sqrt{\sin^2(\alpha) - \cos^2(\theta)}}{\cos(\theta)}$$

伪码如下所示（Miller 和 Goldman 1992）：

```
// Assuming that Abs(Dist(cone.vertex, plane)) < epsilon...
if (Cos(theta) > Sin(alpha)) {
    // Intersection is a point
    intersectionPoint = cone.vertex;
} else if (Cos(theta) == Sin(alpha)) {
    // Intersection is a single line
    line.base = cone.vertex;
    line.direction = Normalize(cone.axis - Cos(theta) * plane.normal);
} else {
    // Intersection is two lines
    u = Normalize(cone.axis - Cos(theta) * plane.normal);
    v = Cross(plane.normal, u);
    sinAlpha = Sin(alpha);
    cosTheta = Cos(theta);
    rVOverRU = Sqrt(sinAlpha * sinAlpha - cosTheta * cosTheta)
               / (1 - sinAlpha * sinAlpha);
    line0.base = cone.vertex;
    line0.direction = Normalize(u + rVOverRU * v);
    line1.base = cone.vertex;
    line0.direction = Normalize(u - rVOverRU * v);
}
```

11.7.5　三角形与圆锥面的相交

设三角形的顶点为P_i，其中$0 \leqslant i \leqslant 2$。圆锥面的顶点为$V$，其轴的方向向量为$\vec{a}$，其轴与外缘的夹角为$\theta$。在大多数的应用中，圆锥面都是锐角的，即$\theta \in (0, \pi/2)$。实际上，本书假设圆锥面都是锐角的，因此$\cos\theta > 0$。圆锥面由点集$X$组成，$X$满足$X - V$与$\vec{a}$之间的夹角为$\theta$。该条件可用代数表达式来表示为

$$\vec{a} \cdot \left(\frac{X - V}{\|X - V\|}\right) = \cos\theta$$

图 11.15 显示了该圆锥面的二维表示。阴影部分表示圆锥面的内侧，将上式中的“＝”换成“≥”就可得到其代数表达式。

为了避免计算$\|X - V\|$中的平方根运算，可对圆锥面方程平方，以得到一个二次方程：

$$\left(\vec{a} \cdot (X - V)\right)^2 = (\cos^2\theta)\|X - V\|^2$$

然而，满足该方程的点集构成一个双锥（double cone）。原来的圆锥面位于平面$\vec{a} \cdot (X - V) = 0$的$\vec{a}$所指向的一面。这个二次方程定义了原圆锥面和基于上述平面的反射。特别地，如果X是这个二次方程的解，那么这个解基于顶点的反射$2V - X$也是一个解。图 11.16 显示了这样的双锥。

为了去掉反射圆锥面，二次方程的解还应该满足$\vec{a} \cdot (X - V) \geqslant 0$。同时，上述二次方程也可以改写为如下的二次方程形式，即$(X - V)^{\mathrm{T}}\mathbf{M}(X - V) = 0$，其中$\mathbf{M} = (\vec{a}\vec{a}^{\mathrm{T}} - \gamma^2 I)$且$\gamma = \cos\theta$。因此，$X$是锐角圆锥面上的点，并满足

$$(X - V)^{\mathrm{T}}\mathbf{M}(X - V) = 0 \quad 且 \quad \vec{a} \cdot (X - V) \geqslant 0$$

1．检测相交

对一些应用程序来说，只检测三角形与圆锥面是否相交而不计算交点是很有用的。例如，由一个点发出的光仅仅照亮位于一个圆锥面内的三角形。知道三角形顶点的颜色是否会由于光的影响而改变是非常有用的。在大多数的图形学应用程序中，如果有一些三角形被照亮，那么所有顶点的颜色都会被计算。知道三角形位于圆锥面内的部分（这是寻找相交问题的答案）并不重要。另一个例子是从被一个圆锥面包围的平截视图中精选三角形，以提高精选的速度。

如果一个三角形与一个圆锥面相交，那么必定是相交于它的一个顶点、一个边点或者一个内点。这里描述的算法使用的检测次序是顶点位于圆锥面内、边与圆锥面相交，以及三角形与圆锥面相交，以提供（满足一定条件时的）提前结束算法的机制。对于许多的三角形都趋向于完全位于圆锥面内的应用程序来说，这种次序是很好的。根据应用程序如何组织其世界数据结构，可能使用其他的次序。

为了检测P_0是否位于圆锥面内，只需检测该点与圆锥面是否位于平面$\vec{a} \cdot (X - V) \geqslant 0$的同一侧，以及该点是否位于双锥内就足够了。虽然检测可以按如下的方式进行

```
D0 = triangle.P0 - cone.V;
AdD0 = Dot(cone.A, D0);
D0dD0 = Dot(D0, D0);
if (AdD0 >= 0 and AdD0 * AdD0 >= cone.CosSqr * D0dD0)
```

```
        triangle.P0 is inside cone;
```

然而如果所有的三角形顶点都位于单锥的外面，那么在边与圆锥面的相交检测中，知道这些顶点位于平面$\vec{a}\cdot(X-V)=0$的哪一侧是非常重要的。按下面的方式进行顶点检测更好一些。术语"在圆锥面之外"是指位于单锥的外面，而不是双锥的外面（一个点可能位于原圆锥面之外，而位于其反射锥面之内）。

```
D0 = triangle.P0 - cone.V;
AdD0 = Dot(cone.A, D0);
if (AdD0 >= 0) {
    D0dD0 = Dot(D0, D0);
    if (AdD0 * AdD0 >= cone.CosSqr * D0dD0) {
        triangle.P0 is inside cone;
    } else {
        triangle.P0 is outside cone, but on cone side of plane;
    }
} else {
    triangle.P0 is outside cone, but on opposite side of plane;
}
```

三角形的全部 3 个顶点都用这种方式来检测。

如果全部 3 个顶点都位于圆锥面之外，那么下一步就检测三角形的边与圆锥面是否相交。考虑边$X(t)=P_0+t\vec{e}_0$，其中$\vec{e}_0=P_1-P_0$且$t\in[0,1]$。如果$\vec{a}\cdot(X(t)-V)\geqslant 0$且$(\vec{a}\cdot(X(t)-V))^2-\gamma^2\|X(t)-V\|^2=0$（$t\in[0,1]$），那么边与单锥相交。第二个条件是一个二次方程，$Q(t)=c_2t^2+2c_1t+c_0=0$，其中$c_2=(\vec{a}\cdot\vec{e}_0)^2-\gamma^2\|\vec{e}_0\|^2$，$c_1=(\vec{a}\cdot\vec{e}_0)(\vec{a}\cdot\vec{\Delta}_0)-\gamma^2\vec{e}_0\cdot\vec{\Delta}_0$，并且$c_0=(\vec{a}\cdot\vec{\Delta}_0)^2-\gamma^2\|\vec{\Delta}_0\|^2$，其中$\vec{\Delta}=P_0-V$。对$Q(t)$求根的定义域取决于顶点位于平面的哪一侧。

如果P_0和P_1都位于平面的反面时，那么边与圆锥面不相交。如果P_0和P_1都与圆锥面位于平面的同一侧，那么必须考虑整条边，因此我们需要确定对于$t\in[0,1]$，$Q(t)=0$是否成立。而且，这种检测应该很快，因为我们不需要知道它们相交于何处，只需要知道是否相交。由于两个顶点都位于圆锥面的外面，而且仅当$t=0$和$t=1$时才成立，因而我们已经知道$Q(0)<0$和$Q(1)<0$。为了使上述的二次方程在$[0,1]$内具有一个根，它的图形必须是凹的，因为如果它的图形是凸的，则它的图形将位于连接点$(0,Q(0))$和$(1,Q(1))$的线段的下面。这条线段与$Q=0$不会相交。因此，图形为凹的条件为$c_2<0$。此外，满足局部极大值的t值必须出现在$[0,1]$内。该值为$\hat{t}=-c_1/c_2$。我们可以通过除法运算来直接计算$\hat{t}$；然而，除法是可以避免的。由于$c_2<0$，因此测试$0\leqslant\hat{t}\leqslant 1$等价于$0\leqslant c_1\leqslant -c_2$。存在一个根的最后一个条件为$Q(\hat{t})\geqslant 0$。当该二次方程的判别式为非负时，该条件成立，即$c_1^2-c_0c_2\geqslant 0$。简而言之，当$P_0$和$P_1$位于平面的同一侧时，如果满足下面的条件，那么对应的边与圆锥面相交：

$$c_2<0 \text{ 且 } 0\leqslant c_1\leqslant -c_2 \text{ 且 } c_1^2\geqslant c_0c_2$$

如果P_0位于圆锥面的一侧，而P_1位于另一侧，则Q的定义域可以缩减为$[0,\bar{t}]$，其中$P_0+\bar{t}\vec{e}_0$为边与平面的交点。参数值为$\bar{t}=-(\vec{a}\cdot\vec{\Delta}_0)/(\vec{a}\cdot\vec{e}_0)$。如果该点是$V$，而且它是边与圆锥面惟一的交点，那么粗略地看，这里给出的算法似乎并不能处理这种情形，因为这种算法假设在边线段对应$[0,\bar{t}]$的端点处有$Q<0$。似乎$Q(\bar{t})=0$和$c_2\geqslant 0$能满足出现相交的条件。然而，这种情形的几何图形说明包含该边的直线与圆锥面并不相交。只有当$Q(t)\leqslant 0$时，

才能相交，因此必须满足条件$c_2 < 0$。现在我们来分析当Q在区间$[0,\tilde{t}]$内有根时的情形。与前面的情形一样，$c_2 < 0$是必需的条件，因为$Q(0) < 0$且$Q(\tilde{t}) < 0$。出现局部极大值的t值必须位于定义域 $0 \leqslant \hat{t} \leqslant \tilde{t}$内。为了避免除法运算，可以将其改写为 $0 \leqslant c_1$和$c_2(\vec{a} \cdot \vec{\Delta}_0) \leqslant c_1(\vec{a} \cdot \vec{e}_0)$。二次方程的判别式为非负的条件仍然需要满足。简而言之，当P_0位于圆锥面的一侧，而P_1位于平面的另一面时，如果满足下面的条件，那么对应的边与圆锥面相交：

$$c_2 < 0 \text{ 且 } 0 \leqslant c_1 \text{ 且 } c_2(\vec{a} \cdot \vec{\Delta}_0) \leqslant c_1(\vec{a} \cdot \vec{e}_0) \text{ 且 } c_1^2 \geqslant c_0 c_2$$

最后，如果P_1位于圆锥面的一侧，而P_0位于另一侧，则Q的定义域可以缩减为$[\tilde{t}, 1]$。它的图形也必须是凹的，二次方程的判别式必须是非负的，并且$\hat{t} \in [\tilde{t}, 1]$。当满足下面的条件时，对应的边与圆锥面相交：

$$c_2 < 0 \text{ 且 } c_1 \leqslant -c_2 \text{ 且 } c_2(\vec{a} \cdot \vec{\Delta}_0) \leqslant c_1(\vec{a} \cdot \vec{e}_0) \text{ 且 } c_1^2 \geqslant c_0 c_2$$

三角形的全部 3 条边都用这种方法来检测。

如果全部的 3 条边都位于圆锥面的外面，三角形与圆锥面也可能相交。如果它们相交，则相交所得的曲线是一个位于三角形内部的椭圆。而且，圆锥面的轴与三角形必定相交于这个椭圆的中心。通过计算圆锥面的轴与三角形所在的平面的相交并证明交点位于三角形内，就足以证明三角形与圆锥面相交。但是，当所有 3 个顶点都位于平面的另一侧时，并不需要进行这种检测——由于此时我们已经知道这些顶点是位于平面的哪一侧，因此可以提前退出算法。

三角形的法线是$\vec{n} = \vec{e}_0 \times \vec{e}_1$。如果圆锥面的轴$V + s\vec{a}$与平面$\vec{n} \cdot (X - P_0) = 0$相交，则当$s = (\vec{n} \cdot \vec{\Delta}_0)/(\vec{n} \cdot \vec{a})$时出现交点。交点可以用平面坐标表示为

$$V + s\vec{a} = P_0 + t_0\vec{e}_0 + t_1\vec{e}_1$$

或者

$$(\vec{n} \cdot \vec{\Delta}_0)\vec{a} - (\vec{n} \cdot \vec{a})\vec{\Delta}_0 = t_0(\vec{n} \cdot \vec{a})\vec{e}_0 + t_1(\vec{n} \cdot \vec{a})\vec{e}_1$$

定义$\vec{u} = (\vec{n} \cdot \vec{\Delta}_0)\vec{a} - (\vec{n} \cdot \vec{a})\vec{\Delta}_0$。为了求解$t_0$，在方程的两边在右边叉乘$\vec{e}_1$，然后点乘$\vec{n}$。类似地，为了求解$t_1$，在方程的两边在右边叉乘$\vec{e}_0$，然后点乘$\vec{n}$。结果为

$$t_0(\vec{n} \cdot \vec{a})\|\vec{n}\|^2 = \vec{n} \cdot \vec{u} \times \vec{e}_1 \text{ 和 } t_1(\vec{n} \cdot \vec{a})\|\vec{n}\|^2 = -\vec{n} \cdot \vec{u} \times \vec{e}_0$$

若要该点位于三角形内，必须满足$t_0 \geqslant 0, t_1 \geqslant 0$和（$t_0 + t_1 \leqslant 1$）。比较时可以不进行除法运算，但是要求两种取决于$\vec{n} \cdot \vec{a}$的符号的情形。在代码中，计算了$\vec{n}, \vec{n} \cdot \vec{a}, \vec{n} \cdot \Delta_0, \vec{u}$和$\vec{n} \times \vec{u}$。如果$\vec{n} \cdot \vec{a} \geqslant 0$，那么当$\vec{n} \times \vec{u} \cdot \vec{e}_0 \leqslant 0, \vec{n} \times \vec{u} \cdot \vec{e}_1 \geqslant 0$和$\vec{n} \times \vec{u} \cdot \vec{e}_2 \leqslant (\vec{n} \cdot \vec{a})\|\vec{n}\|^2$时，该点位于三角形内。当$\vec{n} \cdot \vec{a} \leqslant 0$时，上述的不等式都要反转。

2. 寻找相交

上一节的分析可以扩展到精确地将三角形划分为位于圆锥面内的分量和位于圆锥面外的分量。分离它们的曲线可能是一条二次曲线，也可能是一条线段。如果三角形表示为$X(s,t) = P_0 + s\vec{e}_0 + t\vec{e}_1$，其中$s \geqslant 0, t \geqslant 0$且$s + t \leqslant 1$，那么，单锥与三角形的交点由下式确定

$$\vec{a} \cdot (X(s,t) - V) \geqslant 0 \text{ 和 } (\vec{a} \cdot (X(s,t) - V))^2 - \gamma^2\|X(s,t)\|^2 = 0$$

如果三角形的任何一部分都满足线性不等式，那么三角形的定义域将被剪切为一个子集：整个三角形，一个子三角形，或者一个子二次曲线。在该子定义域中，要解决的问题是确定该二次曲线函数值在何处为零。因此，问题简化为二维空间中一个三角形或二次曲线与一个二次对象的相交。要定位零值，就需要精确地求出在前一节中讨论的三角形边的$Q(t)$的根，如果圆锥面经过三角形的内点，就需要确定相交所得到的椭圆。

11.8　平面对象与多项式曲面的相交

在本节中，我们介绍一个平面与一个多项式曲面的相交问题，图 11.53 显示了这样的一个例子。

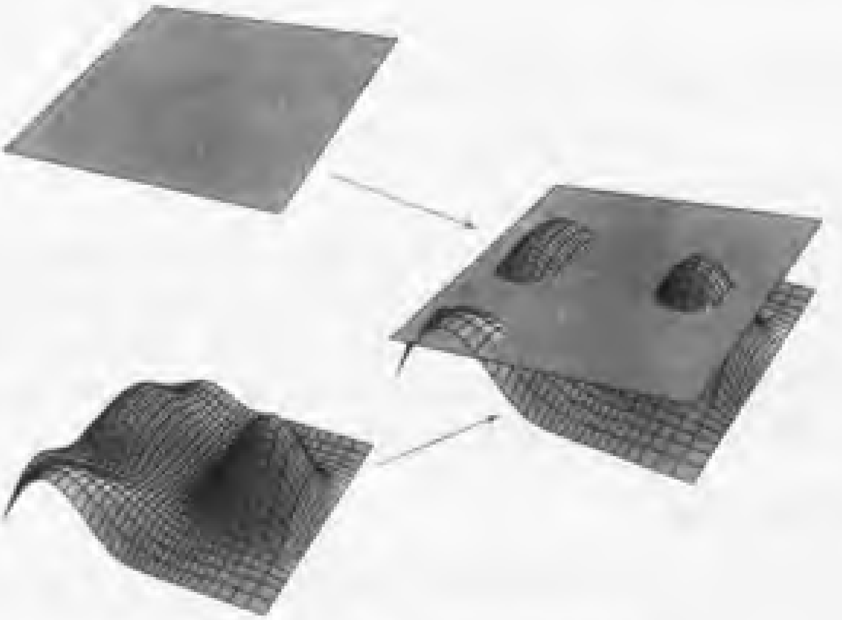


图 11.53　平面与参数形式曲面的相交

非常常用的一种表示多项式曲面的方法是使用有理数参数形式

$$x=\frac{x(s,t)}{w(s,t)}$$

$$y=\frac{y(s,t)}{w(s,t)} \tag{11.27}$$

$$z=\frac{z(s,t)}{w(s,t)}$$

其中多项式$x(s,t)$、$y(s,t)$和$z(s,t)$可能是单项式、贝塞尔函数、B-样条函数或其他的分段多项式基本函数。在这种情形下，有两种一般的方法可用来计算这样的多项式曲面与一个平面的相交。

一种方法是应用一系列的变换（旋转和平移）将相交的平面映射到XY平面。将同样的变换应用于多项式曲面可得

$$x' = \frac{x'(s,t)}{w'(s,t)}$$

$$y' = \frac{y'(s,t)}{w'(s,t)}$$

$$z' = \frac{z'(s,t)}{w'(s,t)}$$

现在的方程$z' = 0$表示在有理多项式曲面的参数空间上的相交。

另一种方法采用另一种方式——将多项式曲面的参数方程代入平面方程

$$ax + by + cz + d = 0$$

如果我们将方程（11.27）中的表达式代入平面方程，可得

$$ax(s,t) + by(s,t) + cz(s,t) + dw(s,t) = 0$$

这是在曲面空间内相交所得曲线的方程。

我们也可以将平面－多项式曲面相交问题看成是一般曲面－曲面相交问题中的一种实例（参见 11.10 节）。由于平面的次数很低，并且平面是平坦的，因而这种方法应该是相对可行的。然而，我们可以更直接地利用其中的一个曲面是一个平面这一事实，并因此得出一个处理这种情形的更高效和更健壮的算法。在文献 Boeing（1997）及 Lee 和 Fredricks（1984）中可以找到两种这样的算法。我们在此提供了后面一种算法。

11.8.1　埃尔米特曲线

我们将要描述的算法可得出一条可用埃尔米特形式来（近似地）表示的相交曲线，因此我们简单地回顾一下埃尔米特表示法。

一条埃尔米特曲线可用其两个端点P_0和P_1，以及两个切线向量$\vec{d}_0$和$\vec{d}_1$来表示。三次埃尔米特基函数为

$$a_0(t) = 2t^3 - 3t^2 + 1$$

$$a_1(t) = -2t^3 + 3t^2$$

$$b_0(t) = t^3 - 2t^2 + t$$

$$b_1(t) = t^3 - t^2$$

埃尔米特基函数显示在图 11.54 中。

这一曲线是上述点和切线向量的一个线性组合，组合参数为上述的（三次）埃尔米特基函数，即：

$$C(t) = a_0(t)P_0 + a_1(t)P_1 + b_0(t)\vec{d}_0 + b_1(t)\vec{d}_1$$

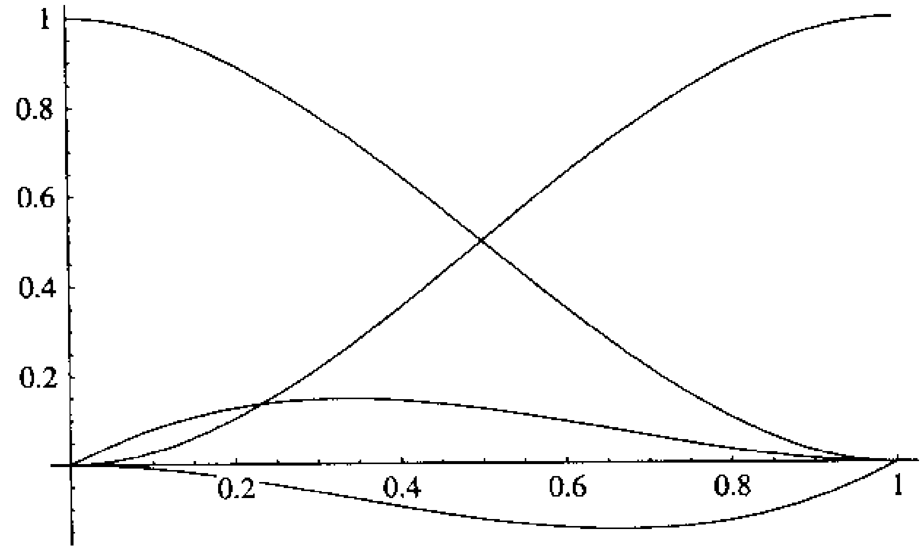


图 11.54　（三次）埃尔米特基函数

图 11.55 中显示了这种曲线的一个例子（由于空间的原因，图中的切线并没有按比例画出；$\vec{d}_0$ 的长度应该是所画长度的$\sqrt{10}$倍，$\vec{d}_1$的长度应该是所画长度的 5 倍）。

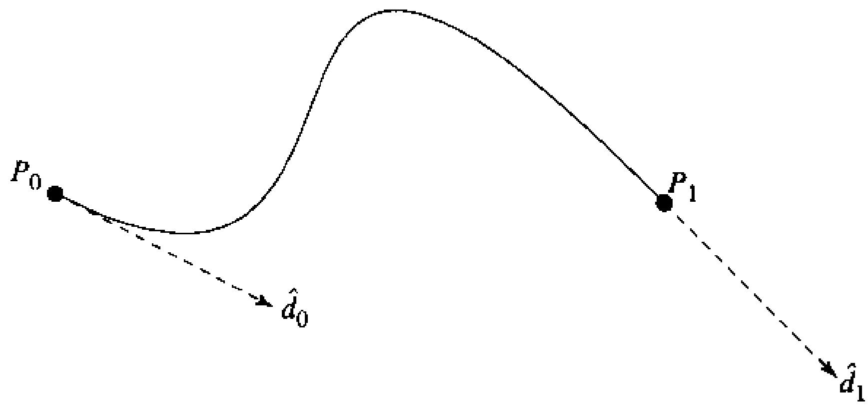


图 11.55　由端点和切线确定的三次埃尔米特曲线

11.8.2　几何定义

我们感兴趣的是一个平面$\mathcal{P}$与一个定义为$S(u, v)$的参数曲面S之间的相交。为了便于讨论，我们假设参数的定义域是 $0 \leqslant u$，$v \leqslant 1$。一个曲面可以看成是$\mathcal{P}$上的许多子片的组合；找到与每一个子片相交得到的曲线，并将这些曲线组合成一条或多条曲线，就得到了与整个曲线的完整相交。用其在参数空间中角的坐标(a, b, c, d)来表示一个子片，其中(a, b)是其左下角的坐标，(c, d)是其右上角的坐标，如图 11.56 所示。

设曲面S与平面$\mathcal{P}$的相交图形为$R(t)$，并设其前像为$p(t) = [\,u(t) \quad v(t)\,]$。我们希望找到一条能够近似地满足方程$R(t) = S(p(t))$的曲线。该算法由两个递归方法组成，两个递归方法用于计算近似于$R(t)$和$p(t)$（的线段）的三次埃尔米特函数。注意，埃尔米特曲线段（它们的组合完全包围相交得到的曲线）是定义在三维空间上的曲线。其近似条件如下：

- 如果其在三维空间中的对应图像位于平面$\mathcal{P}$上，则$p(t)$的近似可认为是“充分的”。
- 如果其位于对应的参数空间线段的三维空间图像的指定容差之内，则$R(t)$的近似可认为是“充分的”。

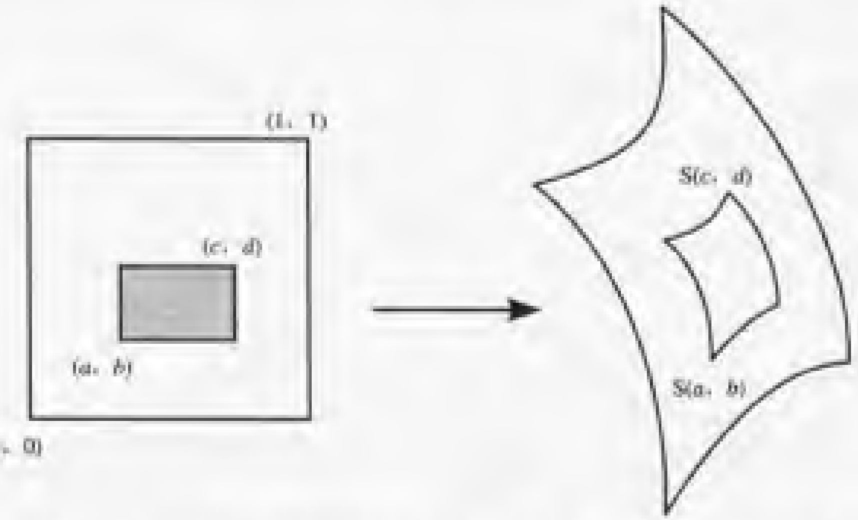


图 11.56　参数空间中的一个子片映射到小片上的一个拓扑三角区域（Lee 和 Fredricks 1984）

11.8.3　计算曲线

回顾一下，在我们使用埃尔米特曲线时，$R(t)$的切线向量为

$$R'(t) = S_u(p(t))u'(t) + S_v(p(t))v'(t) \tag{11.28}$$

其中$S_u(p(t))$和$S_v(p(t))$分别是位于参数t处的关于u和v的偏导数，而u'和v'分别是位于参数t处的p的切线向量的分量。

由于$R(t)$上的所有点都必须位于平面$\mathcal{P}$上，因此必须满足下式

$$R'(t) \cdot \hat{n} = 0$$

其中$\hat{n}$是$\mathcal{P}$的单位法线向量。代入方程（11.28），可得

$$\left(S_u(p(t)) \cdot \hat{n}\right) + \left(S_v(p(t)) \cdot \hat{n}\right)$$

在点$p(t)$处，该方程给出了$R(t)$的切线。通过估计$u'(t)$或$v'(t)$之一的值，并求解另一个值，可以计算出长度的（初始）估值。通过确定$p(t)$的中点的图像位于$\mathcal{P}$内，可以使估值变得更精确。

图 11.57 和图 11.58 分别显示了曲线$R(t) = \{P_0, P_1, \vec{d}_0, \vec{d}_1\}$和$p(t) = \{(u_0, v_0), (u_1, v_1), (u'_0, v'_0), u'_1, v'_1]$（切线向量也没有按比例画出）。

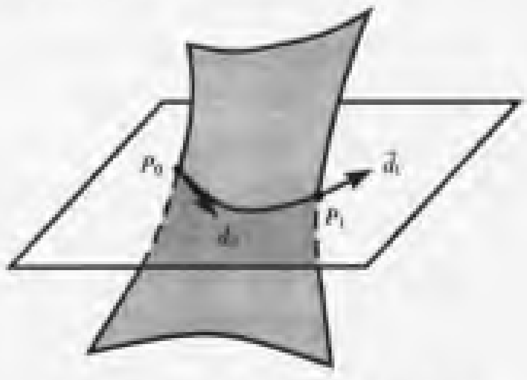


图 11.57　三维空间中的相交曲线 $R(t)$

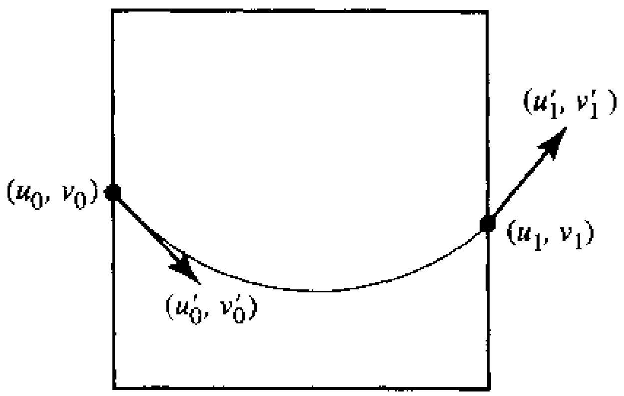


图 11.58 参数空间中的相交曲线 $p(t)$

11.8.4 算法

该算法由 3 部分组成，前面的两部分都是递归的。第一部分将曲面S细分为子片，直到找不到相交为止，或者直到找到一个沿着子片的边刚好与$\mathcal{P}$相交两次的子片。将这样找到的每一个子片，以及三维交点和每一条相交的边，传递到第二个递归算法，该算法计算相交得到的曲线（段）。

伪码如下所示：

```
Find(S, P, a, b, c, d) {
    // Compute intersections along borders, if any
    hits = ComputeIntersectionWithPlane(u0, v0, u1, v1, border0, border1);

    if (hits == 0) {
        return;
    }

    // Check for two hits, on different borders
    if (hits == 2 and border0 != border1) {
        // Get the 3D points of the intersections
        p0 = S(u0, v0);
        p1 = S(u1, v1);

        ComputeCurve(S, P, u0, v0, u1, v1, p0, p1);
    } else {
        // Split the subpatch in half, alternating
        SplitSubPatch(S, a, b, c, d, parm, whichDirection);

        // Recursively solve each half
        if (whichDirection == uDir) {
            Find(S, P, a, b, parm, d);
            Find(S, P, parm, b, c, d);
        } else {
            Find(S, P, a, b, c, parm);
            Find(S, P, a, parm, c, d);
        }
```

```
        }
    }
```

算法的第二部分计算相交得到的曲线，递归地改善精度。正如 Lee 和 Fredricks（1984）所指出的，递归是二叉的——即检查相交所得的曲线的中点，如果它不满足收敛条件，则在该点处将曲线分成两部分，算法对每一部分进行递归——然而建议将曲线分成更多的部分，这样效果可能更好。

伪码如下所示：

```
ComputeCurve(S, P, u0, v0, u1, v1, p0, p1)
{
    Calculate:
        u0' = u'(u0, v0), v0' = v'(u0, v0) and
        u1' = u'(u1, v1), v1' = v'(u1, v1)
        as described in the text

    // Compute tangent vectors for R(t)
    d0 = sSubU(u0, v0) * u'(u0, v0) + sSubV(u0, v0) + v'(u0, v0);
    d1 = sSubU(u1, v1) * u'(u1, v1) + sSubV(u1, v1) + v'(u1, v1);

    Calculate midpoint (uMid, vMid) of
      parameter-space curve p: { (u0, v0), (u1, v1), (u0', v0'), (u1', v1') }

    // Calculate 3D image of p(uMid, vMid)
    pMid = S(uMid, vMid);
    recurse = false;

    // Check if we've met convergence criteria, both 2D and 3D
    if (pMid is not ''close enough'' to plane P) {
        // Intersect curve to find split point
        pMid = IntersectCurve(S, P, uMid, vMid);

        recurse = true;
    } else if (pMid is not ''close enough'' to 3D curve { p0, p1, d0, d1 } {
        recurse = true;
    }

    if (recurse) {
        // Recursively call function on split curve
        ComputeCurve(S, P, u0, v0, p0, uMid, vMid, pMid);
        ComputeCurve(S, P, uMid, vMid, pMid, u1, v1, p1);
    } else {
        // Found curve
        curve = { p0, p1, d0, d1 }
    }
}
```

有必要对函数“IntersectCurve”做一点解释：该函数用一个与点($uMid, vMid$)相邻的等参曲线$S(u_0, v)$或$S(u, v_0)$来计算与平面$\mathcal{P}$的交点，返回新的（$uMid$，$vMid$）。

算法的第三部分接受所有的埃尔米特（子）曲线，并将它们组合在一起，以得到合适

的完整的相交曲线。注意，子曲线的端点应该完全匹配，这样提供的子片的相连边的相交才能得到与平面的统一相交。

11.8.5 实现要点

正如 Lee 和 Fredricks 所指出的，如下情形是可能的：一个小片与平面相交而它的边都不与平面相交。实现上述算法的一种合理的方法是，对每一个小片都计算其封闭的（一个轴对齐或有向有界箱的）体积，并检测这个箱子与平面的相交。如果没有找到相交，则该小片本身与平面也不相交；否则，递归分解小片并检测有界箱与平面的相交，直到封闭的体积不再与平面相交或者找到一个边界与平面相交的子小片为止。如果小片是用贝塞尔或 B—样条基函数来表示的，那么封闭和分解都可被高效地实现。

11.9 二次曲面之间的相交

两个二次曲面之间可能相交为多种不同的曲线形式：

- 一个点——例如，两个球面相接触于它们的极点。
- 一条直线——例如，两个平行的圆柱面，仅仅沿着它们的边相接触。
- 一条曲线——例如，两个部分相交的球面相交于一个圆。
- 两条曲线——例如，一个平面与一个双锥面相交于两条抛物线曲线。
- 一个四次非平面空间曲线——例如，两个圆锥面的相交。

一般来说，前面的 4 种情形是相对简单的，例如，我们在 11.7.3 节中所看到的情形，以及我们将在下面看到的，两个椭球的相交：所有的曲线都是二次曲线段。得到这些二次曲线段的参数表示是可能的，而且这对于许多应用程序来说是非常有利的（例如，高效地对相交曲线着色）。Miller 和 Goldman（1995）描述了检测两个二次曲面何时相交于一条二次曲线段的几何算法，如果相交存在，该算法还可用于计算相交。

正如 Levin（1976，1979，1980）和 Sarraga（1983）所证明的，精确地用参数形式来表示非平面空间曲线也是可能的。Miller（1987）提出了一种处理所谓的自然二次曲面的方法——球面、圆锥面和圆柱面——这种算法完全以几何（而不是代数）技术为基础，并得出了曲线的参数表示。

11.9.1 一般相交问题

在本节中，我们将讨论处理二次曲面相交的一般方法。有两种不同的方法可以表示二次曲面。第一种方法是代数方法，利用关于x，y和z的一般隐含方程来表示曲面：

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gx + 2Hy + 2Jz + K = 0$$

特殊类型的曲面（球面、圆锥面等）可以通过系数值的不同“模式”来表示（例如，对于球面有$A = 1$，$B = 1$，$C = 1$和$K = -1$）。

第二种方法，二次曲面还可表示为矩阵形式

$$PQP^{\mathrm{T}} = 0$$

其中

$$P = [x \quad y \quad z \quad 1]$$

并且

$$\mathbf{Q} = \begin{bmatrix} A & D & F & G \\ D & B & E & H \\ F & E & C & J \\ G & H & J & K \end{bmatrix}$$

两个二次曲面$\mathbf{Q}_0$和$\mathbf{Q}_1$的线性组合：

$$\mathbf{Q} = \mathbf{Q}_0 - \lambda \mathbf{Q}_1$$

定义了一束二次曲面，一般称为二次曲面束。当$\mathbf{Q}_0$和$\mathbf{Q}_1$相交时，两个二次曲面束中每一个二次曲面都与原来的两个二次曲面相交于相同的曲线。

1. 代数方法

代数方法利用了一种 Miller（1987）称之为"不直观却显而易见"的性质——即曲面束中至少存在一个约束曲面。在相交问题中，约束曲面很有用，因为它们可被容易地（并且精确地）参数化。

约束曲面可被表示为一个有理多项式函数或者三角几何形式。三角几何表示法用它们的规范位置和方向来表示曲面，因此任意位置和方向的曲面都被变换到它们的规范位置，并在规范位置计算相交，然后用逆操作将相交变换回原来的位置。一个位于规范位置的圆柱面可用三角几何形式表示为

$$\begin{aligned} x &= r\cos(\theta) \\ y &= r\sin(\theta) \\ z &= s \end{aligned} \tag{11.29}$$

而半角为α的圆锥面可表示为

$$\begin{aligned} x &= s\tan(\alpha)\cos(\theta) \\ y &= s\tan(\alpha)\sin(\theta) \\ z &= s \end{aligned} \tag{11.30}$$

给定两个二次曲面Q_0和Q_1，确定其中一个曲面为参数化曲面——用其多项式项来表示相交得到的曲线。相交得到的曲线可定义为

$$a(t)s^2 + b(t)s + c(t) = 0 \tag{11.31}$$

对于任意特定的相交，可用下面的方法来计算a，b和c：

（1）将一个曲面变换到确定为参数化曲面的本地坐标空间中。

（2）将方程（11.29）或方程（11.30）（使用合适的）代入变换所得的二次曲面的隐含方程中。

（3）对结果进行代数运算，以得到方程（11.31）的形式。对圆柱面来说，上述步骤 2 中的代入可得

$$a(t) = C$$

$$b(t) = 2Er\sin(t) + 2Fr\cos(t) + 2J$$

$$c(t) = Ar^2\cos^2(t) + Br^2\sin^2(t) + 2Dr^2\sin(t)\cos(t) + 2Gr\cos(t) + 2Hr\sin(t) + K$$

而对圆锥面，可得

$$\begin{aligned} a(t) &= A\tan^2(\alpha)\cos^2(t) + B\tan^2(\alpha)\sin^2(t) + C + 2D\tan^2(\alpha)\sin(t)\cos t \\ &\quad + 2E\tan(\alpha)\sin(t) + 2F\tan(\alpha)\sin(t) \\ b(t) &= 2G\tan(\alpha)\cos(t) + 2H\tan(\alpha)\sin(t) + 2J \\ c(t) &= K \end{aligned} \tag{11.32}$$

对于任意特定的相交构形都必须确定其参数t的范围。为了实现这一点，我们就必须确定曲线的临界点——即曲线拐向自身、与自己相交或者趋于无穷的位置。使用代数分析来寻找使s趋于无穷的t值，以及判别式为零的位置。得到的t－空间的分区定义了曲线的有效范围。然后我们可以遍历t－空间的有效分区，并将t代入方程（11.32）的合适形式中。通过求解这个二次曲线方程（方程（11.31））可以求得s。我们自该点可得到成对的值(s_i, t_i)。这些成对的值可被代入到合适的参数方程（对于圆锥面和圆柱面）中，以得到形式为$[x \quad y \quad z]$的点。这些点然后再被变换到世界空间中。

由于操作都相对简单，因此都很容易处理。正如 Miller 所指出的，这类（隐含的）三次或二次代数方程都“对系数的微小变换非常敏感”。图形学中常用的变换（包括将一个二次曲面变换到另一个进行实际的相交处理的空间中的变换）一般都会引起系数的变化。而且，求得交点后，还要将其（逆）变换，系数用来确定哪一个曲面作为参数化曲面。最后，我们必须在最终的计算结果中采用一个趋近于零的容差，然而，这样做是非常困难的，因为将该容差与空间距离联系起来是很困难的。由于这些原因，Miller 建议使用一种几何方法。

Dupont，Lazard 和 Petitjean（2001）的最近的研究表明，Levin 的方法的一些数值缺陷可被有效的减少。首先，我们定义一些项。

一个矩阵为 **P** 的二次曲面$\mathcal{P}$的符号差是一个有序数对(p, q)，其中p和q分别为 **P** 的正的和负的特征值的数目。Dupont 等人指出，如果 **P** 的符号差为(p, q)，则$-\mathbf{P}$的符号差为(q, p)，即使与 **P** 和$-\mathbf{P}$相关的曲面是相同的。因此，他们将符号差定义为(p, q)，其中$p \geqslant q$。**P** 的左上角 3×3 子矩阵表示为$\mathbf{P}_u$，被称为 **P** 的主子矩阵，其行列式叫做主子行列式。这个主子行列式非常重要，因为对于一些简单的约束二次曲面来说，其值为零。

二次曲面的规范形式为

$$\sum_{i=1}^{p} a_i x_i^2 - \sum_{i=p+1}^{r} a_i x_i^2 + \xi = 0$$

或者

$$\sum_{i=1}^{p} a_i x_i^2 - \sum_{i=p+1}^{r} a_i x_i^2 + x_{r+1} = 0$$

其中$a_0 > 0, \forall i, \xi \in [0, 1]$且$p \leqslant r$。简单的约束二次曲面的规范形式如表 11.2 所示。

表 11.2　规范的简单约束二次曲面的参数表示法（Dupont, Lazard 和 Petitjean 2001）

曲面类型	规范方程($a_i > 0$)	参数表示法($\mathbf{X} = [x_1, x_2, x_3], u, v \in \mathbb{R}$)
直线	$a_1x_1^2 + a_2x_2^2 = 0$	$\mathbf{X}(u) = [0, 0, u]$
平面	$x_1 = 0$	$\mathbf{X}(u, v) = [0, u, v]$
双平面	$a_1x_1^2 = 0$	$\mathbf{X}(u, v) = [0, u, v]$
平行平面	$a_1x_1^2 = 1$	$\mathbf{X}(u, v) = [\frac{1}{\sqrt{a_1}}, u, v]$ $\mathbf{X}(u, v) = [\frac{1}{-\sqrt{a_1}}, u, v]$
相交平面	$a_1x_1^2 - a_2x_2^2 = 0$	$\mathbf{X}(u, v) = [\frac{u}{\sqrt{a_1}}, \frac{u}{\sqrt{a_2}}, v]$ $\mathbf{X}(u, v) = [\frac{u}{\sqrt{a_1}}, -\frac{u}{\sqrt{a_2}}, v]$
双曲抛物面	$a_1x_1^2 - a_2x_2^2 - x_3 = 0$	$\mathbf{X}(u, v) = [\frac{u+v}{2\sqrt{a_1}}, \frac{u-v}{2\sqrt{a_2}}, uv]$
抛物柱面	$a_1x_1^2 - x_2 = 0$	$\mathbf{X}(u, v) = [u, a_1u^2, v]$
双曲柱面	$a_1x_1^2 - a_2x_2^2 = 0$	$\mathbf{X}(u, v) = [\frac{1}{2\sqrt{a_1}}(u + \frac{1}{u}), \frac{1}{2\sqrt{a_2}},$ $(u + \frac{1}{u}), v]$

Levin 的处理两个二次曲面$\mathcal{P}$ 和$\mathcal{Q}$的方法可以总结为：

（1）在$\mathcal{P}$和$\mathcal{Q}$的束中找到一个约束二次曲面。这可通过确定$\mathcal{Q}$和$\mathcal{R}(\lambda) = \mathcal{P} - \lambda\mathcal{Q}$（其中$\lambda$为$(\mathcal{R}_u(\lambda))$的解）的类型来实现。

（2）计算将$\mathcal{R}$变换为规范形式的正交变换**T**。在该坐标系中，$\mathcal{R}$具有参数形式（如表 11.2 所示）。在坐标系中计算$\mathcal{P}$的$\mathbf{P}' = \mathbf{T}^{-1}\mathbf{P}\mathbf{T}$，并考虑

$$\mathbf{X}^{\mathrm{T}}\mathbf{P}' = a(u)v^2 + b(u)v + c(u) = 0$$

（3）求解上述方程，用u来表示v。确定定义这些解的u的定义域——这是判别式$b^2(u) - 4a(u)c(u) \geq 0$ 的值域。将u所表示的v的解代入X，得到一个参数式$\mathcal{P} \cap \mathcal{Q} = \mathcal{P} \cap \mathcal{R}$（在$\mathcal{R}$为规范的坐标系中）。

（4）将参数式相交公式变换回世界空间中，即$TX(u)$。

Dupont 等人指出了可能出现数值问题的 3 个地方：

- 在步骤 1 中，λ是一个三次多项式的根，可能需要进行嵌套的立方和平方根运算。
- 在步骤 2 中，**T**的系数涉及高达四次方根的嵌套运算。
- 在步骤 3 中，必须进行求解步骤 2 中的多项式方程的平方根运算。

他们建议了多种对 Levin 的方法的改进方法，以避免上述问题。首先，他们仅考虑$\mathbb{P}^3$上的二次方程——实投影三维空间。Levin 的方法成立是因为存在足够多的规范的二次曲线，在步骤 3 中将它们代入可得到一个具有一个未知数的二次方程，其判别式为关于另一个变量的四次多项式。Dupont 等人提供了一组投影空间的参数化表示，除符号差为（3，1）（椭球面、双曲面和椭球抛物面）的曲面外，它们都共享这一性质。表 11.3 显示了这一点。

表 11.3　投影二次曲面的参数表示法（Dupont, Lazard 和 Petitjean 2001）

符　号　差	规范方程($a_i > 0$)	参数表示法($\mathbf{X} = [x_1, x_2, x_3, x_4] \in \mathbb{P}^3$)
(4,0)	$a_1x_1^2 + a_2x_2^2 + a_3x_3^2 + a_4x_4^2 = 0$	$\mathcal{Q} = \emptyset$
(3,1)	$a_1x_1^2 + a_2x_2^2 + a_3x_3^2 + a_4x_4^2 = 0$	$\det\mathcal{Q} < 0$
(3,0)	$a_1x_1^2 + a_2x_2^2 + a_3x_3^2 = 0$	$\mathcal{Q}$ is a point

续表

符号差	规范方程($a_i > 0$)	参数表示法($\mathbf{X} = [x_1, x_2, x_3, x_4] \in \mathbb{P}^3$)
(2,2)	$a_1x_1^2 + a_2x_2^2 - a_3x_3^2 - a_4x_4^2 = 0$	$\mathbf{X}(u, v) = [\frac{ut+vw}{\sqrt{a_1}}, \frac{uw-vt}{\sqrt{a_2}}, \frac{ut-vw}{\sqrt{a_3}}, \frac{uw+vt}{\sqrt{a_4}}]$
(2,1)	$a_1x_1^2 + a_2x_2^2 - a_3x_3^2 = 0$	$\mathbf{X}(u, v) = [\frac{u^2+v^2}{2\sqrt{a_1}}, \frac{u^2-v^2}{2\sqrt{a_2}}, \frac{uv}{\sqrt{a_3}}, wt]$
(2,0)	$a_1x_1^2 + a_2x_2^2 = 0$	$\mathbf{X}(u, v) = [0, 0, u, v], u, v \in \mathbb{P}^1$
(1,1)	$a_1x_1^2 - a_2x_2^2 = 0$	$\mathbf{X}(u, v) = [\frac{u}{\sqrt{a_1}}, \frac{u}{\sqrt{a_2}}, v, w]$, $\mathbf{X}(u, v) = [\frac{u}{\sqrt{a_1}}, -\frac{u}{\sqrt{a_2}}, v, w], (u, v, w) \in \mathbb{P}^2$
(1,0)	$a_1x_1^2 = 0$	$\mathbf{X}(u, v) = [0, u, v, w], (u, v, w) \in \mathbb{P}^2$

这种方法减少了必须在曲面束中搜索曲面$\mathcal{R}$的次数；当且仅当$\mathcal{P}$和$\mathcal{Q}$的符号为（3，1）时，才需要进行这种搜索。

他们的方法的另一个不同之处是他们采用了一个不同的定理：如果两个二次曲面$\mathcal{P}$和$\mathcal{Q}$都具有符号差（3，1），并且具有多于两个的交点，那么存在一个有理数λ，使得$\mathcal{P}-\lambda\mathcal{Q}$不具有符号差（3，1）。这能保证$\mathcal{P}$和$\mathcal{R}$具有有理系数，并且，“在大多数情形中”，可用普通的浮点算术来计算λ。

最后，他们采用高斯消元法来求解（在步骤 2 中）将$\mathcal{R}$变换到规范坐标系中的二次方程（Lam 1973)。Levin 的方法在这一步中使用正交变换，与此相比，Dupont 等人的方法被“充分地”简化了。

他们的方法可概述为：

（1）在$\mathcal{P}$和$\mathcal{Q}$的束中寻找满足$\det \mathbf{R} \geq 0$ 的二次曲面$\mathcal{R}$（或者，寻找零维的相交）。这可通过检测$\det \mathcal{Q}$实现：如果它是非负的，则设$\mathcal{R} = \mathcal{Q}$；如果存在一个$\lambda$满足$\det(\mathbf{P} - \lambda\mathbf{Q}) \geq 0$，那么设$\mathcal{R} = \mathcal{P} - \lambda\mathcal{Q}$。否则，相交是零维的。

（2）如果$\mathcal{R}$的符号差是（4，0）或（3，0），则相交是零维的。否则，计算将$\mathcal{R}$变换为规范形式的变换$\mathbf{T}$。然后计算$\mathbf{P}' = \mathbf{T}^{\mathrm{T}}\mathbf{P}\mathbf{T}$。

（3）求解$\mathbf{X}^{\mathrm{T}}\mathbf{P}'\mathbf{X} = 0$，并确定$(u, v) \in \mathbb{P}^1$的有效定义域。将$(w, t)$，$wt$或$w$代入用$(u, v)$来表示的$\mathbf{X}$（取决于不同的情形）。如果对某些$(u, v)$的值，$a$，$b$和$c$都为零，那么将$\mathbf{X}$中的这些值替换$(u, v)$，并设$(w, t) \in \mathbb{P}^1$。这就得到了$\mathcal{P}$和$\mathcal{Q}$在$\mathcal{R}$的坐标系内相交的一个参数表示。

（4）将解变换回世界空间中，即$\mathbf{TX}(u, v)$。

这种算法的结果是二次曲面$\mathcal{P}$和$\mathcal{Q}$的相交图形的一个显式的参数表示。所有的系数为$\mathbb{Q}[\sqrt{a_1}, \sqrt{a_2}, \sqrt{a_3}, \sqrt{a_4}]$，其中$a_i$为$\mathbf{T}^{\mathrm{T}}\mathbf{R}\mathbf{T}$的对角线上的系数。

2. 几何方法

几何方法之所以叫做几何方法，是因为在这种方法中，二次曲面是用点、向量和说明曲面类型的数量来表示的。例如，球面用球心和半径来表示。除了具有更多的数值优点外，这种方法与在图形库或应用程序接口中常用对象的定义更匹配。

在讨论球面、圆锥面和圆柱面与平面的相交问题时（11.7 节），我们已经见过几何方法。正如在本节的简介部分中所说明的，两个二次曲面之间的相交图形可能为多种不同的曲线形式：

- 一个点——例如，两个球面相接触于它们的极点。
- 一条直线——例如，两个平行的圆柱面，仅仅沿着它们的边相接触。
- 一条曲线——例如，两个部分相交的球面相交于一个圆。
- 两条曲线——例如，一个平面与一个双锥相交于两条抛物线曲线。
- 一个四次非平面空间曲线——例如，两个圆锥面的相交。

（1）自然二次曲面相交得到平面二次曲线

当然，任何两个自然二次曲面相交都得到这种类型的相交图形。Miller 和 Goldman（1995）注意到两个自然二次曲面的相交图形可以由一条二次曲线（当然，该曲线位于平面上，并可能退化为由一条直线、多条直线或一个点组成）或者一条四次方非平面空间曲线组成。两个自然二次曲面相交得到的平面二次曲线的构形是非常特殊的情形，也确实只有少数的几种类型，而且利用二次曲面的几何表示方法，它们的相交可以用纯几何方法来求得。这类算法本身由相关的两个二次曲面的类型来说明，类似于平面与自然二次曲面的相交问题的描述方法（参见 11.7.2 节，11.7.3 节和 11.7.4 节）。

我们在前面已经提到，自然二次曲面之间的平面相交的性质类似于前面介绍过的平面与自然二次曲面相交的性质，因此，在表 11.4 中，我们仅仅显示了两个二次曲面相交得到一条二次曲线（或退化成点或直线的形式）的情形。

表 11.4　自然二次曲面之间相交得到平面二次曲线的条件。据 Miller 和 Goldman (1995)

表面对	几何条件	结　果
球面—球面	所有	空，一个切点，或者一个圆
球面—圆柱面	位于圆柱面的轴上的球心	空，一个相切的圆，或者两个圆
球面—圆锥面	位于圆锥面的轴上的球心	空，一个相切的圆，一个圆和一个顶点，或者两个圆
圆柱面—圆柱面	平行轴	空，一条切线，或者两条直线
	相交轴且半径相等	两个椭圆
圆柱面—圆锥面	重合轴	两个圆
	轴相交于与圆锥面的顶点相距 $d = \frac{r}{\sin\alpha}$ 的一个点	两个椭圆（相同或者相对的半锥）或者一个椭圆和一条切线
圆锥面—圆锥面	平行轴，相同的半角	椭圆，共用相切的母线
	重合轴	两个圆或者一个顶点
	轴相交于点 I，该点满足条件 $d_1 \sin\alpha_1 = d_2 \sin\alpha_2$，其中 d_i 是顶点 i 到 I 的距离	成对的圆锥曲线，或者一条切线加一条圆锥曲线，或者如果顶点重合则为 1–4 条直线的不同组合

对于自然二次曲面之间相交为非平面曲线的构形，它们的相交是四次非平面空间曲线。当然，这些情形也可以用纯几何方法来计算（Miller 1987），并且每一种算法都是特定类型的。分别全面介绍自然二次曲面相交为平面和非平面曲线的论文（Miller 和 Goldman 1995；Miller 1987）包括 44 页，而且说明平面相交的部分本身又是两篇更长的、提供了细节的技术报告（Miller 和 Goldman 1993a，1993b）的摘要。即使有这么多的文章，包括论

述平面—自然二次曲面的相交问题的论文（Miller 和 Goldman 1992），也仅仅讨论了自然二次曲面之间的相交。当然，你也可以说，这是到目前为止最有用的内容。然而，在任何情形中，任意二次曲面之间的相交都必须用如下方法之一来处理：Levin 方法（1976，1979，1980）、Levin 方法的改进方法（Dupont, Lazard 和 Petitjean 2001）或者作为一般的曲面—曲面相交问题来处理。

（2）二次曲面相交得到非平面曲线

处理这种情形的方法有长久的发展历史，一般分别采用不同的处理方法，来处理二次曲面之间相交得到非平面四次空间曲线的不同情形，我们在此仅仅提供这种方法的要点，对于其细节和实现，读者可以参考 Miller 的详尽论述。

选择一个曲面作为参数曲面，将另一个曲面变换到前一个曲面的本地空间，并用一个由两个曲面组成的曲面束来参数表示相交的曲线。然而，不同对象的表示方法不同：对于参数曲面，使用与坐标无关的参数表示法，而对于其他的曲面，使用隐含方程的指定对象类型的形式。由于参数曲面的参数定义是以具有几何意义的实体为基础的，因此方程 11.31 中的函数$a(t)$，$b(t)$和$c(t)$具有更直接的几何解释。

我们用与坐标无关的项来改写方程（11.29）和方程（11.30）。将圆柱面定义为是由基点B，半径r和轴$\hat{w}$所组成的对象。我们还需要另外的两个向量$\hat{u}$ 和$\hat{v}$ 来参数化表示圆柱面——它们与B和$\hat{w}$一起构成一个正交基底。我们的方程变成

$$P(s,t) = B + r(\cos(t)\hat{u} + \sin(t)\hat{v}) + s\hat{w} \tag{11.33}$$

对圆锥面进行相似的处理，可得

$$P(s,t) = B + s(\tan(\alpha)(\cos(t)\hat{u} + \sin(t)\hat{v}) \tag{11.34}$$

球面、圆柱面和圆锥面的用类型来说明的隐式方程如下所示：

$$\text{球面：}\quad (P-B)\cdot(P-B) - r^2 = 0 \tag{11.35}$$

$$\text{圆柱面：}\quad (P-B)\cdot(P-B) - ((P-B)\cdot\hat{w})^2 - r^2 = 0 \tag{11.36}$$

$$\text{圆锥面：}\quad ((P-B)\cdot\hat{w})^2 - \cos^2(\alpha)(P-B)\cdot(P-B) = 0 \tag{11.37}$$

Miller 的方法的基本步骤可简述如下：

① 选择（可能）相交的两个曲面中的一个作为参数曲面。

② 将参数曲面的参数形式（例如，方程（11.33）或方程（11.34））代入其他曲面的隐式方程（方程（11.35），（11.36）和（11.37）之一）。

③ 处理得到的方程，直到成为$a(t)s^2 + b(t)s + c(t) = 0$的形式。

正如 Miller 所指出的，得到的方程中的函数a，b和c将具有“显而易见的几何解释”。

11.9.2 椭球面

本节将描述如何计算椭球面之间的相交问题。我们会讨论如下的典型问题：检测相交但不计算交点集。此外，我们还会说明如何确定一个椭球面是否完全包含另一个椭球面。后者的方法与 7.5.3 节中描述的处理椭圆的方法具有相同的理论基础。测试两个椭球面$\mathcal{E}_0$和$\mathcal{E}_1$的相交问题时，我们需要回答的精确问题包括：

- $\mathcal{E}_0$和$\mathcal{E}_1$是否相交?
- $\mathcal{E}_0$和$\mathcal{E}_1$是否分离（是否存在一个平面，使得这两个椭球面分别位于这个平面的两边）?
- $\mathcal{E}_0$是否完全包含在$\mathcal{E}_1$内，或者$\mathcal{E}_1$是否完全包含在$\mathcal{E}_0$内?

寻找交点集更复杂。我们讨论了许多的方法。在本节中，椭球面$\mathcal{E}_i$用二次方程$Q_i(X) = X^{\mathrm{T}}\mathrm{A}_i X + B_i^{\mathrm{T}} X + c_i = 0$来定义，其中$\mathrm{A}_i$是一个 3×3 正值矩阵，$B_i$是一个 3×1 向量，$c_i$是一个数量，$X_i$是一个表示椭球面上的点的 3×1 向量。由于A是正值，因此$Q_i(X) < 0$定义椭球面的内部，而$Q_i(X) > 0$定义椭球面的外部。

1. 检测相交

这里的分析以二次函数的阶层曲面为基础。A.9.1 节提供了关于函数的阶层集合的讨论。由$Q_0(x,y,z) = \lambda$所定义的所有阶层曲面都是椭球面，惟一的例外是当λ为最小值（负值）时，该方程定义的是一个点，它是所有阶层曲面的中心点。由$Q_1(x,y,z) = 0$所定义的阶层曲面是一般与Q_0的许多阶层曲面都相交的曲面。其中的问题是如何对$\mathcal{E}_1$上的任一点求得Q_0一阶层值的最小值λ_0和Q_0一阶层值的最大值λ_1。如果$\lambda_1 < 0$，则$\mathcal{E}_1$完全包含于$\mathcal{E}_0$内。如果$\lambda_0 > 0$，则$\mathcal{E}_0$和$\mathcal{E}_1$分离，或者$\mathcal{E}_1$包含$\mathcal{E}_0$。否则，$0 \in [\lambda_0, \lambda_1]$，并且$\mathcal{E}_0$与$\mathcal{E}_1$相交。图 7.7，7.8 和 7.9 显示了二维的情形，但是也完全适用于三维的情形。

这个问题可以明确地表述为具有约束条件的优化问题，可用拉格朗日乘子法（A.9.3 节）来求解。用约束条件$Q_1(X) = 0$来优化$Q_0(X)$。定义$F(X,t) = Q_0(X) + tQ_1(X)$。关于$X$分量微分，可得$\vec{\nabla} F = \vec{\nabla} Q_0 + t\vec{\nabla} Q_1$。关于$t$微分，可得$\partial F/\partial t = Q_1$。设$t$一导数等于零，得到约束条件$Q_1 = 0$。设$X$一导数等于零，得到约束条件$\vec{\nabla} Q_0 + t\vec{\nabla} Q_1 = \vec{0}$。在几何意义上，这说明梯度是平行的。

由于有$\vec{\nabla} Q_i = 2\mathrm{A}_i X + B_i$，因此

$$\vec{0} = \vec{\nabla} Q_0 + t\vec{\nabla} Q_1 = 2(\mathrm{A}_0 + t\mathrm{A}_1)X + (B_0 + tB_1)$$

得到求解X的公式

$$X = -\frac{1}{2}(\mathrm{A}_0 + t\mathrm{A}_1)^{-1}(B_0 + tB_1) = \frac{1}{\delta(t)} Y(t)$$

其中$\mathrm{A}_0 + t\mathrm{A}_1$是一个关于$t$的线性多项式的矩阵，而且其行列式$\delta(t)$是一个关于$t$的三次多项式。$Y(t)$是关于$t$的三次多项式。将其代入$Q_1(X) = 0$，可得

$$Y(t)^T \mathrm{A}_1 Y(t) + \delta(t) B_1^T Y(t) + \delta(t)^2 C_1 = 0$$

这是一个关于t的六次多项式。可以计算它的根，计算其对应的X值，并计算$Q_0(X)$的值。最小值和最大值分别存储为λ_0和λ_1，并可应用上述的与零的比较式。

2. 寻找相交

椭球面的二次方程可被写为关于z的二次多项式（该多项式的系数为x和y）的函数：$Q_0(x,y,z) = \alpha_0(x,y) + \alpha_1(x,y)z + \alpha_2(x,y)z^2$和$Q_1(x,y,z) = \beta_0(x,y) + \beta_1(x,y)z + \beta_2(x,y)z^2$。使用 A.2 节讨论的消元法，当且仅当贝祖判别式为零时，这两个方程具有普通的z根，

$$R(x,y) = (\alpha_2\beta_1 - \alpha_1\beta_2)(\alpha_1\beta_0 - \alpha_0\beta_1) - (\alpha_2\beta_0 - \alpha_0\beta_2)^2 = 0 \quad (11.38)$$

多项式$R(x,y)$的次数最大为 4。如果(x,y)是$R(x,y) = 0$的解，则普通的z根为

$$z=\frac{\alpha_2\beta_0-\alpha_0\beta_2}{\alpha_1\beta_2-\alpha_2\beta_1}$$

如果方程（11.38）有解，则必定至少有一个解接近于原点。这是一个具有约束条件的最小值问题：基于约束条件$R(x,y)=0$，求x^2+y^2的最小值。应用拉格朗日乘子法（A.9.3节），定义$F(x,y,t)=x^2+y^2+tR(x,y)$。导数为

$$(F_x,F_y,F_t)=(2x+tR_x,2y+tR_y,R)$$

其中的变量下标指示关于这些变量的偏导数。方程$F_t=0$刚好导出约束条件$R=0$。由方程$F_x=0$ 和$F_y=0$可得到$2x+tR_x=0$和$2y+tR_y=0$。消去t，可得到另一个多项式方程

$$S(x,y)=yR_x-xR_y=0 \tag{11.39}$$

多项式$S(x,y)$的最大次数也是 4。

现在我们得到了两个具有两个未知数的方程$R(x,y)=0$和$S(x,y)=0$。每一个多项式都可写成关于y的多项式，其系数为关于x: $R(x,y)=\sum_{i=0}^{4}\alpha_i(x)y^i$和$S(x,y)=\sum_{i=0}^{4}\beta_i(x)y^i$的多项式。这两个关于$y$的多项式的贝祖矩阵是一个 4×4 的矩阵$\mathbf{M}=[\mathbf{M}_{ij}]$，其中

$$\mathbf{M}_{ij}=\sum_{k=\max(4-j,4-i)}^{\min(4,7-i-j)}w_{k,7-i-j-k}$$

$0\leqslant i\leqslant 3$ 和 $0\leqslant j\leqslant 3$，且$w_{i,j}=\alpha_i\beta_j-\alpha_j\beta_i$，其中 $0\leqslant i\leqslant 4$ 和 $0\leqslant j\leqslant 4$。其扩展形式为

$$\mathbf{M}=\begin{bmatrix} w_{4,3} & w_{4,2} & w_{4,1} & w_{4,0} \\ w_{4,2} & w_{3,2}+w_{4,1} & w_{3,1}+w_{4,0} & w_{3,0} \\ w_{4,1} & w_{3,1}+w_{4,0} & w_{2,1}+w_{3,0} & w_{2,0} \\ w_{4,0} & w_{3,0} & w_{2,0} & w_{1,0} \end{bmatrix}$$

α_i和β_i的最大次数为 $4-i$。$w_{i,j}$的最大次数为 $8-i-j$。贝祖判别式为$D(x)=\det(\mathbf{M}(x))$，这是一个关于x的最大次数为 16 的多项式。

计算$D(x)=0$的根。对每一个根$\bar{x}$，计算$f(y)$的系数及$R(\bar{x},y)=0$的y根。如果$\bar{y}$是这样一个根，则测试数对$(\bar{x},\bar{y})$，以确保$S(\bar{x},\bar{y})=0$。如果满足该条件，则我们找到了这两个椭球面的一个交点。如果该交点是一个孤立点，则这两个椭圆相切于该点。如果$\nabla Q_0(\bar{x},\bar{y})$和$\nabla Q_1(\bar{x},\bar{y})$互相平行，则该点是孤立的。只需检验是否满足一个简单的条件，即两个梯度向量的叉积是否为零。如果该点不是孤立的，则交集将由一条封闭曲线构成。可以使用一种微分方程的求解方法来遍历该曲线：

$$\frac{dx}{dt}=R_y(x,y),\ \frac{dy}{dt}=-R_x(x,y),\ (x(0),y(0))=(\bar{x},\bar{y}) \tag{11.40}$$

向量(R_x,R_y)是$R=0$所定义的阶层曲线的法线，因此向量$(R_y,-R_x)$是该阶层曲线的切线。微分方程则可以用来沿着切线向量遍历该曲线。

上一种算法的主要问题是，计算一个 16 次多项式的数值根可能成为一个病态问题。另一种方法是使用迭代搜索，建立一个微分方程系统，该系统允许沿着一个椭球面搜索它与另一个椭球面的交点。这种搜索将找到一个交点或者证明它们不相交。

从椭球面$\mathcal{E}_0$上的一个点X_0开始，因此$Q_0(X_0)=0$。如果$Q_1(X_0)=0$，我们就找到了一个交点。如果$Q_1(X_0)<0$，那么X_0位于$\mathcal{E}_1$内。我们的做法是，沿着第一个椭球面的切线方向

增加Q_1的值，直到零。在空间上，Q_1的最大增幅的方向为$\vec{\nabla}Q_1$。该向量是椭球面$\mathcal{E}_1$的法线，但一般与椭球面$\mathcal{E}_0$并不相切。通过减去$\vec{\nabla}Q_0$的贡献，该向量必须投影到$\mathcal{E}_0$的相切空间中。在Q_1上具有最大增幅的位于$\mathcal{E}_0$上的路径可由下式局部确定

$$\frac{dX}{dt}=\vec{\nabla}Q_1-\frac{\vec{\nabla}Q_1\cdot\vec{\nabla}Q_0}{\|\vec{\nabla}Q_0\|^2}\ \vec{\nabla}Q_0,\ \ X(0)=X_0$$

在$Q_1(X_0)>0$的情形中，切线方向必须反转，使得Q_1能尽可能快地减小到零。这种情形的微分方程为

$$\frac{dX}{dt}=-\vec{\nabla}Q_1+\frac{\vec{\nabla}Q_1\cdot\vec{\nabla}Q_0}{\|\vec{\nabla}Q_0\|^2}\ \vec{\nabla}Q_0,\ \ X(0)=X_0$$

不论椭球面是否相交，遍历最终都将得到一个使梯度平行的点。在这种情形中，微分方程的右式将减小为零向量。右式向量的长度可以作为数值计算方法的结束条件。另一个问题是，由于数值误差，数值计算方法将从老位置得到一个新位置，新位置可能不在第一个椭球面上。可以利用一种校验方法来纠正新的位置，使其位于第一个椭球面上。修正的数值可以用来进行数值求解方法中的下一次迭代。

一旦在某一时间$T>0$找到一个点$X_1=X(T)$满足$Q_1(X_1)=0$，就可以利用方程（11.40）的二维空间阶层曲线遍历。然而，直接在三维空间中遍历相交所得的曲线也是可能的。该曲线的一条切线向量同时垂直于$\vec{\nabla}Q_0$和$\vec{\nabla}Q_1$。要求解的方程系统为

$$\frac{dX}{dt}=\vec{\nabla}Q_0\times\vec{\nabla}Q_1,\ \ X(0)=X_1$$

不论使用哪一个微分方程系统来遍历相交曲线，都必须设置某些类型的停止条件，用来检测遍历已经到达遍历的起始点X_0。

11.10 多项式曲面之间的相交

在本节中，我们将简要地讨论计算两个多项式曲面的相交问题，并从论述这一困难但重要的问题的丰富文献资料中挑选了一些，供读者参考。

在许多的计算机图形学应用中——建模、生成有限元网格、指定工具路径、科学视觉、界面和特征检测，等等——经常遇到曲面－曲面相交问题（SSI）。在边界表示几何建模应用中，曲面－曲面相交问题显然是非常重要的，而且代数曲面和 NURBS 曲面的相交可以看成是在几何和实体建模集成系统中需要考虑的基本问题。图 11.59 显示了两个相交于两个封闭的环的B－样条曲面。图 11.60 显示了在其中一个曲面的参数空间中的相交曲线。

在过去的几十年中，有大量的工作用于研究与曲面－曲面相交的相关问题。最近的调查可见于 Patrikalakis（1993），Pratt 和 Geisow（1986）和 Hoffman（1989）。曲面－曲面相交算法一般可划分为 4 种类型：细分方法、格子评测、解析方法和步进方法。还提出了一些集成上述方法中的几种的混合方法。

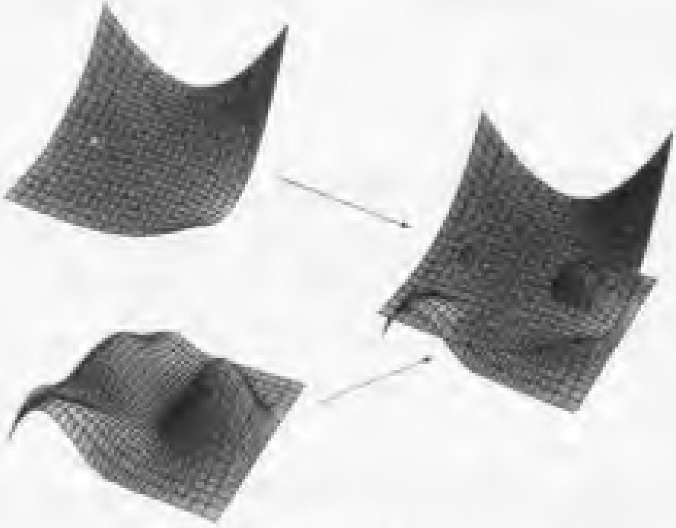


图 11.59 两个 B 样条曲面的相交

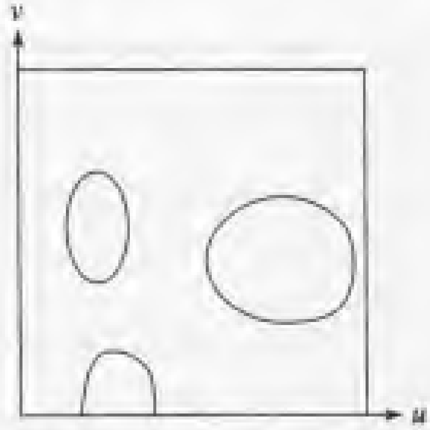


图 11.60 曲面参数空间中的相交曲线

11.10.1 细分方法

这种方法的核心思想是（递归地）将问题分解为更小和更简单的问题。一个简单的例子是递归地将曲面分解为双线性小片，然后再处理这些小片的相交。不断进行细分，直到满足某些判断条件，每一条曲线在满足这些条件的点上拼接在一起，形成一条或多条相邻的曲线。一般地，细分控制是基于控制多边形的几何性质来进行的——例如，不断缩小的性质、凸包，等等（Lane 和 Riesenfeld 1980；Lasser 1986）。在有限范围内，细分方法收敛于实际的相交曲线，但是趋向于产生大量的数据并且运行速度很慢。通过限制细分的层次以减少这类问题，可以增加速度，但是这样存在一些风险，即可能丢失一些小特性，比如环或者被丢失或者错误地连接孤立体。

11.10.2 格子评测

在这种方法中，曲面相交问题简化为一系列的更低阶的几何形状的相交问题，比如曲

线一曲面相交（Rossignac 和 Requicha 1987）。与细分方法一样，格子方法趋向于很慢，也存在关于丢失环和孤立体的问题。

11.10.3 解析方法

这种方法试图找到相交所得曲线的一种明确表示。只有在有限的情形中——相交所得曲线是低阶曲线（Sederberg 1983；Sarraga 1983）和（或）限于特殊曲线或者特殊类型的曲面（Piegl 1989；Miller 1987）——这种方法才可行。Miller 和 Goldman（1992，1995）使用一种几何方法（相对于严格的代数方法而言）来实现这种方法。

11.10.4 步进方法

这种方法又称为曲线跟踪，步进方法的基本思想如下（Hoffman 1989，206）：

在相交的一个点P处构建一条局部的近似曲线。例如，该曲线相切于P。通过沿着该近似曲线步进一段指定的距离，我们可以得到下一个曲线点的估计位置，然后基于该位置利用迭代方法来优化结果。

跟踪 / 步进方法似乎是用得最广泛的方法（Barnhill 和 Kersey 1990；Farouki 1986；Bajaj 等 1989）。其中的原因（Krishnan 和 Manocha 1997）是它（相对地）易于实现并具有通用性（这种方法可以处理偏移和混合之类的情形，这类情形对 CAD 应用来说特别重要）。这种方法可能引起两个重要的问题。首先，相交所得曲线的起始点必须是确定的。正如我们已经看到的，相交所得曲线可能是边界曲线段或者封闭环，而且还可能出现孤立体之类的退化。边界曲线段的起始点可用曲面相交方法来确定（Sederberg 和 Nishita 1991）。封闭环的检测证明了更多问题；大多数方法使用高斯映射来包围和简单地分解曲面，直到“满足证明不存在环的充分条件”（Krishnan 和 Manocha 1997）。

跟踪方法的第二个主要问题是：一旦起始点被确定，大多数算法使用牛顿法的变体来进行实际的跟踪。然而，在实际操作中，步长必须保持为很小，以避免出现所谓的分量跳跃（component jumping）。但是这样做的结果是，这种方法可能非常慢。Krishnan 和 Manocha（1997）最近所做的工作已经很好地研究了这类问题。

11.11 轴分离方法

在三维空间中，轴分离的概念类似于 7.7 节中的二维轴分离的概念。如果存在一条直线，两个固定的凸物体在这条直线上的投影不相交，则这两个固定的凸物体不相交。用于运动凸物体的方法，以及检测相交和寻找相交的方法，都可以用与二维空间相同的方法来表示。

11.11.1 固定凸多面体的分离

在二维空间中，我们的工作集中于凸多边形。在三维空间中，我们将主要研究凸多面体，然而，必须特别注意两个多面体实际上都是平面多边形的情形。在进行一对凸多面体的分离检测时，仅需要考虑有限的方向向量集。这类似于与凸多边形相关的二维问题，但

是我们并不提供这一论断在三维空间中的证明。然而，在直观上，这类似于二维空间中的凸多边形的情形。如果这两个凸多面体仅仅相接触而不相交，则接触方式为面一面接触、面一边接触、面一顶点接触、边一边接触、边一顶点接触或者顶点一顶点接触中的一种。获得这些相交类型的可能方向的集合包括多边体的面的法线向量，以及分别属于两个多面体的两条边的叉积所产生的向量。

设C_j（$j=0,1$）是一个凸多面体，其顶点为$\{V_i^{(j)}\}_{i=0}^{N_j-1}$，边为$\{\vec{e}_i^{(j)}\}_{i=0}^{M_j-1}$，面为$\{F_i^{(j)}\}_{i=0}^{L_j-1}$。设它的面都是平面凸多边形，当从该多面体的外面观察面时，这些凸多边形的顶点是按逆时针方向排序的。指向外面的法线向量可以与每一个面一起存储，作为面的方向。我们假设可以查询每一个面的法线和顶点，并假设可以查询每一条边的顶点。

检测两个凸多面体（两者都具有正的体积$L_j \geq 4$）的相交问题的伪码类似于二维空间中的直接实现：

```
bool TestIntersection(ConvexPolyhedron C0, ConvexPolyhedron C1)
{
    // test faces of C0 for separation
    for (i = 0; i < C0.L; i++) {
        D = C0.F(i).normal;
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if (max1 < min0 || max0 < min1 )
            return false;
    }

    // test faces of C1 for separation
    for (j = 0; j < C1.L; j++) {
        D = C1.F(j).normal;
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if ( max1 < min0 || max0 < min1)
            return false;
    }

    // test cross products of pairs of edges
    for (i = 0; i < C0.M; i++) {
        for (j = 0; j < C1.M; j++) {
            D = Cross(C0.E(i), C1.E(j));
            ComputeInterval(C0, D, min0, max0);
            ComputeInterval(C1, D, min1, max1);
            if (max1 < min0 || max0 < min1)
                return false;
        }
    }

    return true;
}

void ComputeInterval(ConvexPolyhedron C, Point D, float& min, float& max)
{
```

```
        min = Dot(D, C.V(0));  max = min;
        for (i = 1; i < C.N; i++) {
            value = Dot(D, C.V(i));
            if (value < min) min = value; else max = value;
        }
    }
```

在三维空间中，存在一个与二维空间中寻找凸多边形的极值点的一种较好的算法类似的算法。给定n个顶点，寻找极端点的可能时间为$O(\log n)$（Kirkpatrick 1983；Dobkin 和 Kirkpatrick 1990）。O'Rourke（1998）提供了这种算法的一个写得非常好的说明。然而，对于n较小或者其中的多面体具有很多的对称性（典型的情形为定向有界箱）的应用来说，用于实现、测试和调试该算法的时间可能是不平衡的，特别是，与直接的$O(n)$算法中的常数相比，如果$O(\log n)$算法中的常数足够大，则这种情形更加明显。

当两个凸多面体都是同一平面内的凸多边形时，前面列出的伪码TestIntersection存在问题。在这种情形中，一般的法线向量并不是分离方向。任何两条边的叉积也是平面的一个法线向量，因此它不能分离这两个多边形。实际上，必须使用二维空间中的凸多边形处理算法来处理三维空间中的共面凸多边形。在三维空间中，我们所需的就是多边形所在平面上的向量和垂直于相关的边的向量。如果$\vec{n}$是一个平面的法线，$\vec{e}$是多边形的一个边向量，那么$\vec{n} \times \vec{e}$是一个可能的分离方向。伪码为

```
bool TestIntersection(ConvexPolygon C0, ConvexPolygon C1)
{
    // test normal of C0 for separation
    D = C0.normal;
    ComputeInterval(C0, D, min0, max0);
    ComputeInterval(C1, D, min1, max1);
    if (max1 < min0 || max0 < min1)
        return false;

    Point N0xN1 = Cross(C0.normal, C1.normal);
    if (N0xN1 != 0) {
        // polygons are not parallel
        // test normal of C1 for separation
        D = C1.normal;
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        if (max1 < min0 || max0 < min1)
            return false;

        // test cross products of pairs of edges
        for (i = 0; i < C0.M; i++) {
            for (j = 0; j < C1.M; j++) {
                D = Cross(C0.E(i), C1.E(j));
                ComputeInterval(C0, D, min0, max0);
                ComputeInterval(C1, D, min1, max1);

                if (max1 < min0 || max0 < min1)
                    return false;
            }
```

```
            }
        } else {
            // polygons are parallel (coplanar, C0.normal did not separate)
            // test edge normals for C0
            for (i = 0; i < C0.M; i++) {
                D = Cross(C0.normal, C0.E(i));
                ComputeInterval(C0, D, min0, max0);
                ComputeInterval(C1, D, min1, max1);
                if (max1 < min0 || max0 < min1)
                    return false;
            }

            // test edge normals for C1
            for (i1 = 0; i1 < 3; i1++) {
                D = Cross(C1.normal, C1.E(i));
                ComputeInterval(C0, D, min0, max0);
                ComputeInterval(C1, D, min1, max1);
                if (max1 < min0 || max0 < min1)
                    return false;
            }
        }

        return true;
    }
```

在浮点算术系统中，应该修改多边形法线的叉积与零向量的比较方法，即使用相对误差。选择的阈值epsilon是关于两个法线向量之间夹角的正弦平方的阈值，其基础是恒等式$\vec{n}_0 \times \vec{n}_1 = \|\vec{n}_0\|\|\vec{n}_1\| \sin(\theta)$，其中$\theta$是这两个向量之间的夹角。对epsilon的选择取决于应用程序的实际情形。

```
// comparison when the normals are unit length
Point N0xN1 = Cross(C0.normal, C1.normal);
float N0xN1SqrLen = Dot(N0xN1, N0xN1);
if (N0xN1SqrLen >= epsilon) {
    // polygons are (effectively) not parallel
} else {
    // polygons are (effectively) parallel
}
// comparison when the normals are not unit length
Point N0xN1 = Cross(C0.normal, C1.normal);
float N0xN1SqrLen = Dot(N0xN1, N0xN1);
float N0SqrLen = Dot(C0.normal, C0.normal);
float N1SqrLen = Dot(C1.normal, C1.normal);
if (N0xN1SqrLen >= epsilon * N0SqrLen * N1SqrLen) {
    // polygons are (effectively) not parallel
} else {
    // polygons are (effectively) parallel
}
```

11.11.2　运动凸多面体的分离

这种算法的结构类似于二维空间的同类算法的结构。细节参见 7.7 节。检测两个体积

为正的凸多面体的相交的伪码列出如下。

```
bool TestIntersection(ConvexPolyhedron C0, Point W0, ConvexPolyhedron C1,
 Point W1, float tmax, float& tfirst, float& tlast)
{
    W = W1 - W0;  // process as if C0 stationary, C1 moving
    tfirst = 0;  tlast = INFINITY;

    // test faces of C0 for separation
    for (i = 0; i < C0.L; i++) {
        D = C0.F(i).normal;
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, min0, max0, min1, max1, tfirst, tlast))
            return false;
    }

    // test faces of C1 for separation
    for (j = 0; j < C1.L; j++) {
        D = C1.F(j).normal;
        ComputeInterval(C0, D, min0, max0);
        ComputeInterval(C1, D, min1, max1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, min0, max0, min1, max1,
            tfirst, tlast))
            return false;
    }

    // test cross products of pairs of edges
    for (i = 0; i < C0.M; i++) {
        for (j = 0; j < C1.M; j++) {
            D = Cross(C0.E(i), C1.E(j));
            ComputeInterval(C0, D, min0, max0);
            ComputeInterval(C1, D, min1, max1);
            speed = Dot(D, W);
            if (NoIntersect(tmax, speed, min0, max0, min1, max1, tfirst, tlast) )
                return false;
        }
    }

    return true;
}
```

函数NoIntersect就是在二维问题中所使用的那个函数。对于两个凸多面体都是平面多边形的情形，这里并没有提供完整的伪码，但是可以通过用如下形式的代码段

```
if (max1 < min0 || max0 < min1)
    return false;
```

来替换如下形式的代码段：

```
speed = Dot(D, W);
if (NoIntersect(tmax, speed, min0, max0, min1, max1, tfirst, tlast))
    return false;
```

11.11.3 固定凸多面体的交集

求解两个固定的凸多面体的交集问题是多面体的布尔运算的一种特例。13.6节将提供关于计算布尔运算的一般讨论，特别是关于计算凸多面体相交的讨论。

11.11.4 固定凸多面体的接触集

给定两个运动的凸对象C_0和C_1，它们开始并不相交，它们的运动速度分别为$\vec{w}_0$和$\vec{w}_1$，如果$T>0$是它们第一次相交的时间，那么集合$C_0+T\vec{w}_0=\{X+T\vec{w}_0:X\in C_0\}$和$C_1+T\vec{w}_1=\{X+T\vec{w}_1:X\in C_1\}$就是刚好相触但不贯穿的接触点。正如我们在前面提到的，凸多面体的接触为面－面接触、面－边接触、面－顶点接触、边－边接触、边－顶点接触或者顶点－顶点接触中的一种。这种分析比二维空间的情形要复杂一些，但是基本思想是一样的——要合适地计算相触集就必须知道凸多面体之间的相对方向。我们建议首先阅读7.7节，以便在完成本节之前理解这些思想。

可以修改函数TestIntersection以记录哪一个顶点、边或面投影到投影区间的端点。在第一次接触时，该信息用来确定两个物体是如何相对运动的。如果接触是顶点－顶点、顶点－边或顶点－面接触，那么接触点是惟一的一个点，即一个顶点。如果接触是边－边接触，那么接触一般是一个点，但也可能是一条线段。如果接触是边－面接触，那么接触集是一条线段。最后，如果接触是面－面接触，那么相交集是一个凸多边形。这是最复杂的情形，需要使用一种二维空间的凸多边形相交方法。投影区间的每一个端点都是由一个顶点、一条边或者一个面所产生的。与二维空间中的情形相似，可以用一种两个字符的标记来表示投影类型，并与每一个多面体关联起来。一个字符的标记是V表示顶点投影，E表示边投影，F表示面投影。9种两个字符的标记是VV，VE，VF，EV，EE，EF，FV，FE和FF。第一个字符对应于区间的最小值，第二个字符对应于区间的最大值。存储标记的方便的数据结构、投影区间和投影到区间端点的多面体分量的索引都与在二维问题中所使用的完全一样：

```
Configuration
{
    float min, max;
    int index[2];
    char type[2];
}
```

投影区间是[min, max]。作为一个例子，如果投影类型是VF，index[0]就是投影到最小值的顶点的索引，而index[1]就是投影到最大值的面的索引。

声明了两种构形对象，即关于多面体C_0的Cfg0和关于多面体C_1的Cfg1。用ProjectNormal和ProjectGeneral来替换二维问题中处理每一个可能的分离方向$\vec{d}$的调用了ComputeInterval的代码段。函数ProjectNormal知道正在使用的一个边的法线，只需计算另一个端点的投影。函数ProjectGeneral确定位于两个端点的投影类型。必须针对三维问题来实现模拟函数，然而，它们适用于与面的法线相关的分离检测。成对的边的叉积的分离检测要求用

ProjectGeneral来替换ProjectGeneral的两次调用，因为在这类情形中，并不能保证只有边才投影为区间的端点。

下面列出了处理多面体投影到它的一个面的一条法线上的伪码。由于顶点和边的线性排序，二维情形的代码更简单。如果当前的最小值投影出现两次，则在该点处的值是真正的最小值，并且必定有一条边投影到该点上。在三维情形中，不存在线性排序，因此代码更复杂一些。如果当前的最小值投影出现三次，那么顶点必定是沿着当前考虑的方向的一个投影为区间端点的面的一部分。在该点处，可以安全地返回，没有其他的顶点需要处理。复杂的是，一旦你发现由第三个顶点投影到当前的最小值，就必须确定哪一个面包含这些顶点。还有一种可能的情形是，当前的最小值出现两次，并且是由垂直于当前方向的边所产生的，但是这条边并不一定会投影到关于这个方向的一个端点。当前的最小值出现两次时，不可能提前退出。

```
void ProjectNormal(ConvexPolyhedron C, Point D, int faceindex,
                   Configuration Cfg)
{
    // store the vertex indices that map to current minimum
    int minquantity, minindex[3];

    // project the face
    minquantity = 1;
    minindex[0] = C.IndexOf(C.F(faceindex).V(0));
    Cfg.min = Cfg.max = Dot(D,C.V(minindex[0]));
    Cfg.index[1] = faceindex;
    Cfg.type[1] = 'F';

    for (i = 0; i < C.N; i++) {
        value = Dot(D, C.V(i));
        if (value < Cfg.min) {
            minquantity = 1;
            minindex[0] = i;
            Cfg.min = value;
            Cfg.index[0] = i;
            Cfg.type[0] = 'V';
        } else if (value == Cfg.min && Cfg.min < Cfg.max) {
            minindex[minquantity++] = i;
            if (minquantity == 2) {
                if (C.ExistsEdge(minindex[0], minindex[1])) {
                    // edge is parallel to initial face
                    Cfg.index[0] = C.GetEdgeIndexFromVertices(minindex);
                    Cfg.type[0] = 'E';
                }
                // else:  two nonconnected vertices project to current
                // minimum, the first vertex is kept as the current extreme
            } else if ( minquantity == 3 ) {
                // Face is parallel to initial face.  This face must project
                // to the minimum and no other vertices can do so.  No need
                // to further process vertices.
                Cfg.index[0] = C.GetFaceIndexFromVertices(minindex);
```

```
                Cfg.type[0] = 'F';
                return;
            }
        }
    }
}
```

处理一个多面体在直线上的一般投影的伪码如下：

```
void ProjectGeneral(ConvexPolyhedron C, Point D, Configuration Cfg)
{
    Cfg.min = Cfg.max = Dot(D, C.V(0));
    Cfg.index[0] = Cfg.index[1] = 0;

    for (i = 1; i < C.N; i++) {
        value = Dot(D, C.V(i));
        if (value < Cfg.min)  {
            Cfg.min = value;
            Cfg.index[0] = i;
        } else if (value > Cfg.max) {
            Cfg.max = value;
            Cfg.index[1] = i;
        }
    }

    Cfg.type[0] = Cfg.type[1] = 'V';
    for (i = 0; i < 2; i++)  {
        for each face F sharing C.V(Cfg.index[i])  {
            if (F.normal parallel to D) {
                Cfg.index[i] = C.GetIndexOfFace(F);
                Cfg.type[i] = 'F';
                break;
            }
        }

        if (Cfg.type[i] != 'F')  {
            for each edge E sharing C.V(Cfg.index[i])  {
                if (E perpendicular to D) {
                    Cfg.index[i] = C.GetIndexOfEdge(E);
                    Cfg.type[i] = 'E';
                    break;
                }
            }
        }
    }
}
```

在二维空间中修改过的函数NoIntersect（它接受构形对象而不是投影区间作为参数）完全可以用于处理三维问题。经过这样的修改，函数TestIntersection具有如下的等价形式：

```
bool TestIntersection(ConvexPolyhedron C0, Point W0, ConvexPolyhedron C1, Point W1,
    float tmax, float& tfirst, float& tlast)
```

```
{
    W = W1 - W0;  // process as if C0 stationary, C1 moving
    tfirst = 0;  tlast = INFINITY;

    // test faces of C0 for separation
    for (i = 0; i < C0.L; i++) {
        D = C0.F(i).normal;
        ProjectNormal(C0, D, i, Cfg0);
        ProjectGeneral(C1, D, Cfg1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst, tlast) )
            return false;
    }

    // test faces of C1 for separation
    for (j = 0; j < C1.L; j++) {
        D = C1.F(j).normal;
        ProjectNormal(C1, D, j, Cfg1);
        ProjectGeneral(C0, D, Cfg0);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst, tlast))
            return false;
    }

    // test cross products of pairs of edges
    for (i = 0; i < C0.M; i++) {

        for (j = 0; j < C1.M; j++)  {
            D = Cross(C0.E(i), C1.E(j));
            ProjectGeneral(C0, D, Cfg0);
            ProjectGeneral(C1, D, Cfg1);
            speed = Dot(D, W);
            if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst, t last))
                return false;
        }
    }

    return true;
}
```

FindIntersection伪码具有与函数TestIntersection完全相同的实现，不同的只是如果存在一个相交，则会调用一段额外的代码（其实就是一个循环）。当多面体在时间T相交时，它们都有效地以相对速度运动，可以计算它们的接触集。伪码列出如下。交集是一个多面体，并通过函数的最后一个参数返回。如果交集不为空，则返回值为true。否则，原来运动的凸多面体不相交，并且函数返回false。

```
bool FindIntersection(ConvexPolyhedron C0, Point W0, ConvexPolyhedron C1, Point W1,
    float tmax, float& tfirst, float& tlast, ConvexPolyhedron& I)
{
    W = W1 - W0;  // process as if C0 stationary, C1 moving
    tfirst = 0;  tlast = INFINITY;
```

```
    // test faces of C0 for separation
    for (i = 0; i < C0.L; i++) {
        D = C0.F(i).normal;
        ProjectNormal(C0, D, i,Cfg0);
        ProjectGeneral(C1, D, Cfg1);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst, tlast))
            return false;
    }

    // test faces of C1 for separation
    for (j = 0; j < C1.L; j++) {
        D = C1.F(j).normal;
        ProjectNormal(C1, D, j, Cfg1);
        ProjectGeneral(C0, D, Cfg0);
        speed = Dot(D, W);
        if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst, tlast))
            return false;
    }

    // test cross products of pairs of edges
    for (i = 0; i < C0.M; i++)  {
        for (j = 0; j < C1.M; j++) {
            D = Cross(C0.E(i), C1.E(j));
            ProjectGeneral(C0, D, Cfg0);
            ProjectGeneral(C1, D, Cfg1);
            speed = Dot(D, W);
            if (NoIntersect(tmax, speed, Cfg0, Cfg1, Curr0, Curr1, side, tfirst, tlast))
                return false;
        }
    }

    // compute the contact set
    GetIntersection(C0, W0, C1, W1, Curr0, Curr1, side, tfirst, I);
    return true;
}
```

计算相交的伪码显示如下：

```
void GetIntersection(ConvexPolyhedron C0, Point W0, ConvexPolyhedron C1,
     Point W1, Configuration Cfg0, Configuration Cfg1, int side, float tfirst,
     ConvexPolyhedron I)
{
    if (side == 1) {
        // C0-max meets C1-min
        if (Cfg0.type[1] == 'V') {
            // vertex-{vertex/edge/face} intersection
            I.InsertVertex(C0.V(Cfg0.index[1]) + tfirst * W0);
        } else if (Cfg1.type[0] == 'V') {
            // {vertex/edge/face}-vertex intersection
            I.InsertVertex(C1.V(Cfg1.index[0]) + tfirst * W1);
        } else if (Cfg0.type[1] == 'E')  {
```

```
            Segment E0Moved = C0.E(Cfg0.index[1]) + tfirst * W0;
            if (Cfg1.type[0] == 'E') {
                Segment E1Moved = C1.E(Cfg1.index[0]) + tfirst * W1;
                FindSegmentIntersection(E0Moved, E1Moved, I);
            } else {
                ConvexPolygon F1Moved = C1.F(Cfg1.index[0]) + tfirst * W1;
                FindSegmentPolygonIntersection(E0Moved, F1Moved, I);
            }

    } else if (Cfg1.type[0] == 'E') {
        Segment E1Moved = C1.E(Cfg1.index[0]) + tfirst * W1;
        if (Cfg0.type[1] == 'E')  {
            Segment E0Moved = C0.E(Cfg0.index[1]) + tfirst * W0;
            FindSegmentIntersection(E1Moved, E0Moved, I);
        } else {
            ConvexPolygon F0Moved = C0.F(Cfg0.index[1]) + tfirst * W0;
            FindSegmentPolygonIntersection(E1Moved, F0Moved, I);
        }
    } else {
        // Cfg0.type[1] == 'F' && Cfg1.type[0] == 'F'
        // face-face intersection
        ConvexPolygon F0Moved = C0.F(Cfg0.index[1]) + tfirst * W0;
        ConvexPolygon F1Moved = C1.F(Cfg1.index[0]) + tfirst * W1;
        FindPolygonIntersection(F0Moved, F1Moved, I);
    }
} else if ( side == -1 ) {
    // C1-max meets C0-min
    if (Cfg1.type[1] == 'V') {
        // vertex-{vertex/edge/face} intersection
        I.InsertVertex(C1.V(Cfg1.index[1]) + tfirst * W1);
    } else if (Cfg0.type[0] == 'V')  {
        // {vertex/edge/face}-vertex intersection
        I.InsertVertex(C0.V(Cfg0.index[0]) + tfirst * W0);
    } else if (Cfg1.type[1] == 'E') {
        Segment E1Moved = C1.E(Cfg1.index[1]) + tfirst * W1;
        if (Cfg0.type[0] == 'E') {
            Segment E0Moved = C0.E(Cfg0.index[0]) + tfirst * W0;
            FindSegmentIntersection(E1Moved, E0Moved, I);
        }  else  {
            ConvexPolygon F0Moved = C0.F(Cfg0.index[0]) + tfirst * W0;
            FindSegmentPolygonIntersection(E1Moved, F0Moved, I);
        }
    } else if (Cfg0.type[0] == 'E') {
        Segment E0Moved = C0.E(Cfg0.index[0]) + tfirst * W0;
        if (Cfg1.type[1] == 'E') {
            Segment E1Moved = C1.E(Cfg1.index[1]) + tfirst * W1;
            FindSegmentIntersection(E0Moved, E1Moved, I);
        } else {
            ConvexPolygon F1Moved = C1.F(Cfg1.index[1]) + tfirst * W1;
            FindSegmentPolygonIntersection(E0Moved, F1Moved, I);
        }
```

```
            } else {
                // Cfg1.type[1] == 'F' && Cfg0.type[0] == 'F'
                  // face-face intersection
                  ConvexPolygon F0Moved = C0.F(Cfg0.index[0]) + tfirst * W0;
                  ConvexPolygon F1Moved = C1.F(Cfg1.index[1]) + tfirst * W1;
                  FindPolygonIntersection(F0Moved, F1Moved, I);
            }
        } else {
            // polyhedra were initially intersecting
            ConvexPolyhedron C0Moved = C0 + tfirst * W0;
            ConvexPolyhedron C1Moved = C1 + tfirst * W1;
            FindPolyhedronIntersection(C0Moved, C1Moved, I);
        }
    }
```

处理运动凸对象的函数的名称清晰地说明了它们的功能。

11.12 杂项

本节涵盖了一系列不能归类于本章前面几节的相交问题。

11.12.1 有向有界箱与正交平截体的相交

11.11 节中讨论的轴分离方法可用来确定有向有界箱与正交平截体的相交。这对于精确计算有向有界箱关于视图平截体的体积非常有用。

有向有界箱表示为关于中心C，轴方向$\hat{a}_0$, $\hat{a}_1$和$\hat{a}_2$，以及宽度e_0, e_1和e_2的对称形式。它的轴构成一个右手正交系。假设宽度为正。它的 8 个顶点为$C+\sum_{i=0}^{2}\sigma_i e_i \hat{a}_i$，其中$|\sigma_i|=1$（有 8 种方向可供选择）。6 个箱面的 3 个法线向量为$\hat{a}_i$，其中 $0\le i\le 2$。12 个箱边的 3 个边向量也是上述的向量集合。

正交视图平截体具有原点E。其坐标轴由左向量$\hat{l}$，顶向量$\hat{u}$和方向向量$\hat{d}$所确定。以这种次序排列的上述向量构成一个右手正交系。正交视图平截体在$\hat{d}$方向上的宽度为$[n, f]$，其中$0<n<f$。假设视图平面是近平面$\hat{d}\cdot(X-E)=n$。远平面为$\hat{d}\cdot(X-E)=f$。平截体在近平面上的 4 个角为$E\pm\ell\hat{l}\pm\mu\hat{u}+n\hat{d}$。平截体在远平面上的 4 个角为$E+(f/n)(\pm\ell\hat{l}\pm\mu\hat{u}+n\hat{d})$。6 个平截体面的 5 个法线向量为，近平面和远平面的法线为$\hat{d}$，左面和右面的法线为$\pm n\hat{l}-\ell\hat{d}$，顶面和底面的法线为$\pm n\hat{u}-\ell\hat{d}$。12 个平截体边的 6 个边方向向量为，近平面和远平面上的边为$\hat{l}$和$\hat{u}$，以及$\pm\ell\hat{l}\pm\mu\hat{u}+n\hat{d}$。法线和边向量并不都是单位向量，对于分离轴检测来说，这也不是必需的。

1. 分离轴检测

如果存在一条方向为$\vec{m}$的直线，凸多面体在直线上的投影不相交，则凸多面体不相交。在这种情形中，必定存在一个法线向量为$\vec{m}$的平面分离这两个多面体。给定一条直线，将多面体的顶点投影到这条直线上，以及计算投影得到的封闭区间$[\lambda_{\min}, \lambda_{\max}]$，都是很简单的。确定从两个多面体得到的两个区间是否重叠的比较也很简单。

较困难的问题是选择直线方向的一个有限集合，判断相交或者不相交时，可以只使用

这个集合中的向量来进行分离轴检测。对于凸多面体，可以证明该集合由两个多面体的面的法线，以及分别取自两个多面体的一对边的叉积向量组成，使用该集合就足以进行相交检测。如果多面体i具有F_i个面和E_i条边，那么该集合中向量的总数为$F_0 + F_1 + E_0E_1$。某些边的叉积形成的向量可能为零向量，这种向量并不需要检测。例如，处理两个轴对齐有向有界箱时，就会出现这种情形。而总的向量数目为$3 + 3 + 3\times 3 = 15$，该集合中只有 3 个非零向量。

有向有界箱有$F_0 = 3$和$E_0 = 3$。正交视图平截体有$F_1 = 5$和$E_1 = 6$。需要测试的向量总数为 26。该集合为

$$\{\hat{d}, \pm n\hat{l} - \ell\hat{d}, \pm n\hat{u} - \mu\hat{d}, \hat{a}_i, \hat{l} \times \hat{a}_i, \hat{u} \times \hat{a}_i, (\pm\ell\hat{l} \pm \mu\hat{u} + n\hat{d}) \times \hat{a}_i\}$$

要检测的分离轴都具有形式$E + \lambda\vec{m}$，其中$\vec{m}$位于前面提供的那个集合中。这个箱子的投影顶点具有λ值$\vec{m} \cdot (C - E) + \sum_{i=0}^{2} \sigma_i e_i \hat{a}_i$，其中$|\sigma_i| = 1$。定义$d = \vec{m} \cdot (C - E)$和$R = \sum_{i=0}^{2} e_i |\vec{m} \cdot \hat{a}_i|$。投影区间为$[d - R, d + R]$。

平截体的投影顶点具有λ值$\kappa(\tau_0 \ell \vec{m} \cdot \hat{l} + \tau_1 \mu \vec{m} \cdot \hat{u} + n\vec{m} \cdot \hat{d})$，其中$\kappa \in \{1, f/n\}$和$|\tau_i| = 1$。定义$p = \ell|\vec{m} \cdot \hat{l}| + \mu|\vec{m} \cdot \hat{u}|$。投影区间为$[m_0, m_1]$，其中

$$m_0 = \begin{cases} \frac{f}{n}\left(n\vec{m} \cdot \hat{d} - p\right), & n\vec{m} \cdot \hat{d} - p < 0 \\ n\vec{m} \cdot \hat{d} - p, & n\vec{m} \cdot \hat{d} - p \geqslant 0 \end{cases}$$

以及

$$m_1 = \begin{cases} \frac{f}{n}\left(n\vec{m} \cdot \hat{d} + p\right), & n\vec{m} \cdot \hat{d} + p > 0 \\ n\vec{m} \cdot \hat{d} + p, & n\vec{m} \cdot \hat{d} + p \leqslant 0 \end{cases}$$

对于某些$\vec{m}$的选择，如果两个投影区间$[m_0, m_1]$和$[d - R, d + R]$不相交，则箱子和平截体不相交。如果$d + R < m_0$或者$d - R > m_1$，则这两个区间不相交。没有经过优化的实现将对 26 种情形中的每一种计算d, R, m_0和m_1，并检测任意两种是否相等。然而，优化的实现将存储每一种检测的中间结果，并在后面的计算中使用它们。

2. 缓存中间结果

所有可能的分离轴方向都将在平截体的坐标系统中被有效地处理。即每一个方向都表示为$\vec{m} = x_0\hat{l} + x_1\hat{u} + x_2\hat{d}$，而且其中的系数用于不同的检测。差$C - E$也必须用平截体坐标来表示，即$C - E = \delta_0\hat{l} + \delta_1\hat{u} + \delta_2\hat{d}$。表 11.5 给出了各种系数，其中$0 \leqslant i \leqslant 2$，并且$|\tau_j| = 1$（$0 \leqslant j \leqslant 1$）。

表 11.5　轴分解检测的系数

$\vec{m}$	$\vec{m} \cdot \hat{l}$	$\vec{m} \cdot \hat{u}$	$\vec{m} \cdot \hat{d}$	$\vec{m} \cdot (C - E)$
$\hat{d}$	0	0	1	δ_2
$\pm n\hat{l} - \ell\hat{d}$	$\pm n$	0	$-\ell$	$\pm n\delta_0 - \ell\delta_2$
$\pm n\hat{u} - \mu\hat{d}$	0	$\pm n$	$-\mu$	$\pm n\delta_1 - \mu\delta_2$
$\hat{a}_i$	α_i	β_i	γ_i	$\alpha_i\delta_0 + \beta_i\delta_1 + \gamma_i\delta_2$
$\hat{l} \times \hat{a}_i$	0	$-\gamma_i$	β_i	$-\gamma_i\delta_1 + \beta_i\delta_2$

续表

$\vec{m}$	$\vec{m}\cdot\hat{l}$	$\vec{m}\cdot\hat{u}$	$\vec{m}\cdot\hat{d}$	$\vec{m}\cdot(C-E)$
$\hat{u}\times\hat{a}_i$	γ_i	0	$-\alpha_i$	$\gamma_i\delta_0-\alpha_i\delta_2$
$(\tau_0\ell\hat{l}+\tau_1\mu\hat{u}+n\hat{d})\times\hat{a}_i$	$-n\beta_i+\tau_1\mu\gamma_i$	$n\alpha_i-\tau_0\ell\gamma_i$	$\tau_0\ell\beta_i-\tau_1\mu\alpha_i$	$[-n\beta_i+\tau_1\mu\gamma_i]\delta_0+$ $[n\alpha_i-\tau_0\ell\gamma_i]\delta_1+$ $[\tau_0\ell\beta_i-\tau_1\mu\alpha_i]\delta_2$

为了避免不必要的计算，只在需要时才计算数值$\alpha_i,\beta_i,\gamma_i$和$\delta_i$。只在需要时才计算某些乘积，并且结果被保存供以后使用。这些乘积包括n,ℓ或μ与$\alpha_i,\beta_i,\gamma_i$或$\delta_i$的乘积。包含这些乘积的和或差的一些项也仅在需要时才计算，并且结果被保存供以后使用。这些项包括$n\alpha_i\pm\ell\gamma_i, n\beta_i\pm\mu\gamma_i, \ell\alpha_i\pm\mu\beta_i$和$\ell\beta_i\pm\mu\alpha_i$。

11.12.2　线形对象与轴对齐有界箱的相交

轴对齐有界箱（AABB）是一种矩形平行管，它的每一个面都与一个基底向量垂直。在空间细分问题（比如射线跟踪或碰撞检测）中经常用到这种有界箱。定义这种箱子的一种常用方法是用两个点$P_{min}=[x_{min}\quad y_{min}\quad z_{min}]$和$P_{max}=[x_{max}\quad y_{max}\quad z_{max}]$来表示，如图 11.61 所示。

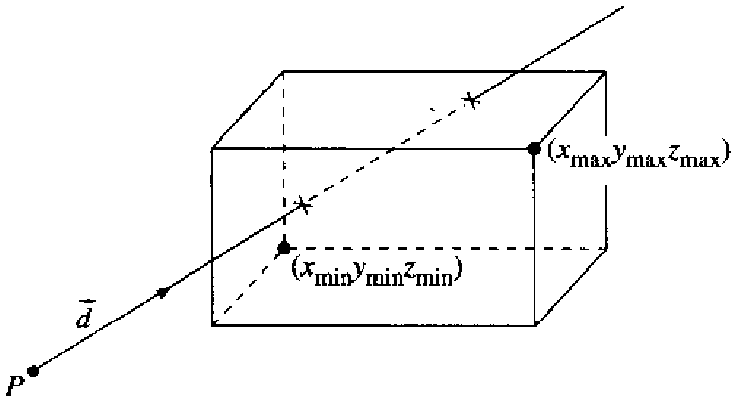


图 11.61　线形对象与轴对齐有界箱的相交

本节讨论寻找一个线形对象与一个轴对齐有界箱的相交问题。可以将一个有界箱看成是由 6 个矩形面组成的，并简单地计算线形对象与每一个矩形的相交。在这类算法中，可以利用所有矩形的边都平行于一个基底向量这一性质，这样可以使这类算法的效率更高。

Kay 和 Kajiya（1986）最早提出了一种叫做“厚板方法”的算法，Haines（1989）采用了这种方法。这种算法的效率更高一些，特别是当线形对象是一条射线时效果更好。而射线是最常见的情形，比如在射线跟踪和选取对象算法中。其基本思想是，相交检测和计算可以被看成是一种裁剪问题。这种方法叫做“厚板”方法是因为可以将轴对齐有界箱看做是 3 个互相垂直的厚板相交，用相对的一对面来定义这些厚板的边界，如图 11.62 所示。

基本的剪裁方法如图 11.63 所示，图中显示了射线$P+t\vec{d}$被平面 YZ 在x_{min}和x_{max}处被裁剪。我们可以看出，在参数值t_0和t_1处的交点可以用下式来计算

$$t_{0,x}=\frac{x_{min}-P_x}{d_x} \qquad (11.41)$$

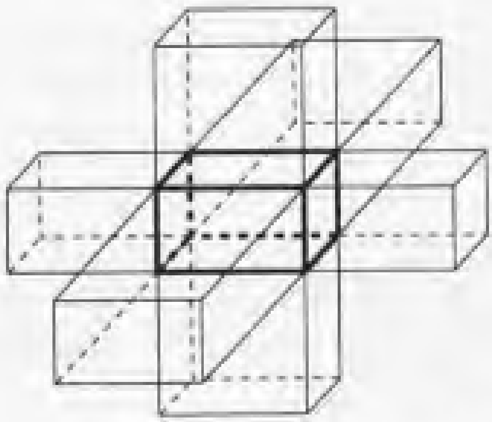


图 11.62　轴对齐有界箱表现为 3 块“厚板”的相交

$$t_{1,x} = \frac{x_{\max} - P_x}{d_x} \tag{11.42}$$

注意，如果射线的端点在厚板的另一侧，并与厚板相交，那么最近的相交将是 t_1 而不是 t_0。

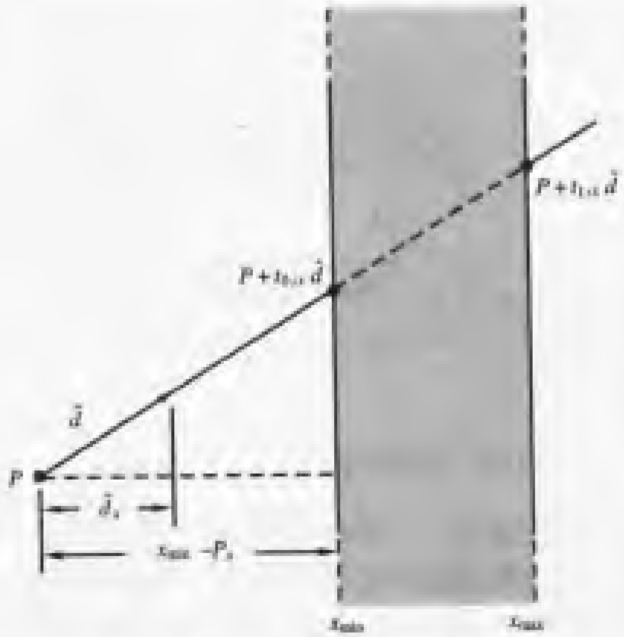


图 11.63　用一块厚板裁剪一条直线

该算法的伪码为

```
boolean RayIntersectAABB(Point P, Vector d, AABB box, float& tIntersect)
{
    tNear = -INFINITY;
    tFar  =  INFINITY;

    foreach pair of planes {(XY_min, XY_max), (XZ_min, YZ_max), (YZ_min, YZ_max)} {
        // Example shown for YZ planes
        // Check for ray parallel to planes
```

```
        if (abs(d.x) < epsilon) {
            //  Ray parallel to planes
            if (P.x < box.xMin || P.x < box.xMax) {
                return false;
            }
        }

        //  Ray not parallel to planes, so find parameters of intersections
        t0 = (box.xMin - P.x) / d.x;
        t1 = (box.xMax - P.x) / d.x;

        //  Check ordering
        if (t0 > t1}) {

             //  Swap them
             tmp = t1;
             t0 = t1;
             t1 = tmp;
         }

         //  Compare with current values
         if (t0 > tNear) {
             tNear = t0;
         }
         if (t1 < tFar) {
             tFar = t1;
         }

         // Check if ray misses entirely
         if (tNear > tFar) {
             return false;
         }
         if (tFar < 0) {
             return false;


         }
     }

     //  Box definitely intersected
     if (tNear > 0) {
         tIntersect = tNear;
     } else {
         tIntersect = tFar;
     }

     return true;
}
```

从上述代码中并不能很清晰地看出这种简单的方法有效的原因，但是从图 11.64 中应该能很清楚地看出来。由于 $t_{near}=t_{0,x}<t_{far}=t_{1,y}$ 和 $t_{near}>0$，所以较低的射线与箱子相交。

因为$t_{near}=t_{0,x}>t_{far}=t_{1,y}$，所以上面的射线与箱子不相交。

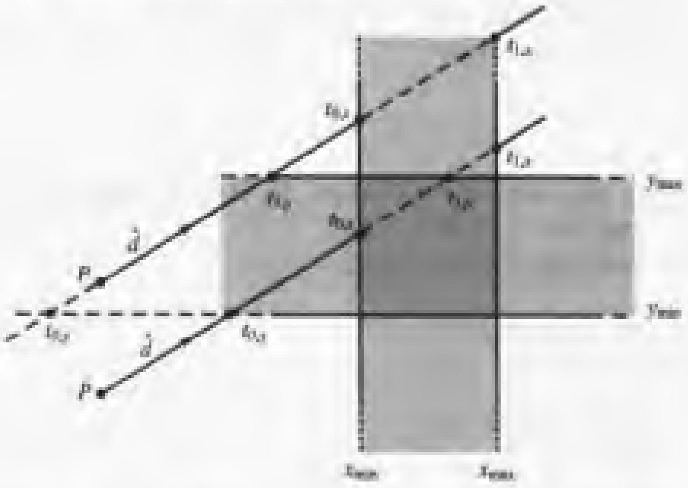


图 11.64 根据适当的厚板裁剪方式来计算射线－AABB 相交

11.12.3 线形对象与有向有界箱的相交

有向有界箱其实就是一种封闭的平行管，它的面和边都不平行于定义它们的坐标系的基底向量。定义它们的一种常用方法是用一个中心点C，一个确定方向和位置的基底向量为$\{\hat{u},\hat{v},\hat{w}\}$的正交系，以及 3 个分别表示半宽、半高和半深的数量来定义，如图 11.65 所示。我们知道，通过将箱子的基底向量作为它所在的空间的基底向量，一个轴对齐有界箱可以严格地用这种方法来定义。然而，最常用的计算轴对齐有界箱的方法是，找到它所包围的几何图元的x，y和z的最大值，这样可以使上一节中使用的轴对齐有界箱的定义更直接一些。

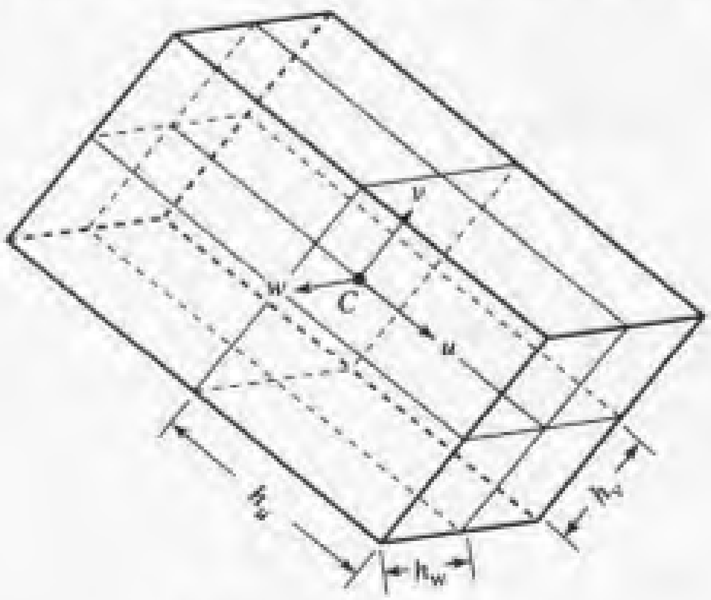


图 11.65 定义一个有向有界箱

由于有向有界箱就是具有不同方向的轴对齐有界箱，因此你可能期望可以用同样的基本算法来计算这种相交，事实上也是如此。Möller和 Haines（1999）描述了一种可以用于有

向有界箱的算法，其实就是上一节介绍的算法的一种改进。

用于有向裁剪的距离计算完全类似于厚板剪裁（参见方程（11.41））：

$$t_{0,u} = \frac{\hat{u} \cdot (C - P) - h_u}{\hat{u} \cdot \hat{d}}$$

$$t_{1,u} = \frac{\hat{u} \cdot (C - P) + h_u}{\hat{u} \cdot \hat{d}}$$

如图 11.66 所示。

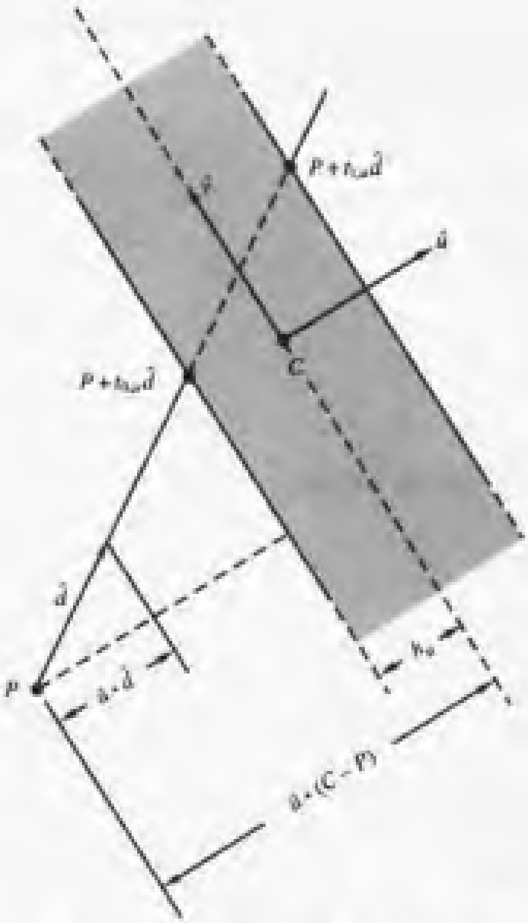


图 11.66 用“定向厚板”进行裁剪

对于与射线相交的情形，在大多数的应用（例如，射线跟踪或交互选取）中，仅要求最近的相交。如果要求两个（可能的）相交，可以修改下面的代码，以同时返回tNear和tFar：

```
boolean RayIntersectOBB(Point P, Vector d, OBB box, float& tIntersect)
{
    tNear = -INFINITY;
    tFar  =  INFINITY;

    foreach (i in {u, v, w}) {
        //  Check for ray parallel to planes
     if (Abs(Dot(d, box.axis[i] < epsilon) {
         //  Ray parallel to planes
```

```
            r = Dot(box.axis[i], box.C - P);
            if (-r - box.halfLength[i] > 0 || -r + box.halfLength[i] > 0) {
                // No intersection
                return false;
            }
        }

        r = Dot(box.axis[i], box.C - P);
        s = Dot(box.axis[i], d);

        // Ray not parallel to planes, so find parameters of intersections
        t0 = (r + box.halfLength[i]) / s;
        t1 = (r - box.halfLength[i]) / s;

        // Check ordering
        if (t0 > t1) {
            // Swap them
            tmp = t0;
            t0 = t1;
            t1 = tmp;
        }

        // Compare with current values
        if (t0 > tNear) {
            tNear = t0;
        }
        if (t1 < tFar) {
            tFar = t1;
        }

        // Check if ray misses entirely
        if (tNear > tFar) {
            return false;
        }
        if (tFar < 0) {
            return false;
        }
    }

    // Box definitely intersected
    if (tNear > 0) {
            tIntersect = tNear;
        } else {
            tIntersect = tFar;
        }

        return true;
    }
```

11.12.4　平面与轴对齐有界箱的相交

本节将讨论一个平面与一个轴对齐有界箱的相交问题。与我们提供的其他相交方法不同，我们在此仅检测是否出现相交。其原因是，轴对齐有界箱本身并不是几何图元，只是用它来包围其他几何图元，一般用于快速排除算法的代码中（比如着色或碰撞检测）：如果不存在平面－轴对齐有界箱相交，我们就不需要检测它所包围的图元；如果确实存在一个平面－轴对齐有界箱相交，那么我们就继续检测平面与它所包围的图元的真正相交。

处理平面－轴对齐有界箱相交的最简单方法就是利用如下的思想：当且仅当轴对齐有界箱的角分别位于平面的两侧时，它们才相交。但是，请注意，我们可能需要检查每一个角，其效率看来并不是很糟糕（检查一个点位于平面的哪一侧并不太费时），除非需要检查与大量的轴对齐有界箱的相交的情形。

对于这个问题，我们采用一般的方法来定义轴对齐有界箱，即用一对点

$$P_{\min} = [x_{\min} \quad y_{\min} \quad z_{\min}]$$

$$P_{\max} = [x_{\max} \quad y_{\max} \quad z_{\max}]$$

来表示，而用与坐标无关的隐含形式来表示平面（参见 9.2.1 节）：

$$\vec{n} \cdot (P - P_0) = 0$$

图 11.67 显示了一个平面与一个轴对齐有界箱的相交。

Möller 和 Haines（1999）观察到如下的结果：如果我们考虑轴对齐有界箱的对角线，对于与该平面的法线最接近平齐的一个对角线，仅需要检测它的两个端点。在二维空间中可以更清楚地看出这一点，如图 11.68 所示。可以采用另一种优化：一旦找到最平齐的对角线，如果最小的点位于平面的正面，我们就能立即得出如下结论，即平面与箱子不相交，因为最大的点也将位于平面的正面。

伪码为

```
boolean PlaneIntersectAABB(Plane plane, AABB box)
{
    // Find points at end of diagonal
    // nearest plane normal
    foreach (dir in (x, y, z)) {
        if (plane.normal[dir] >= 0) {
            dMin[dir] = box.min[dir];
            dMax[dir] = box.max[dir];
        } else {
            dMin[dir] = box.max[dir];
            dMax[dir] = box.min[dir];
        }
    }

    // Check if minimal point on diagonal
    // is on positive side of plane
    if (Dot(plane.normal, dMin) + plane.d) >= 0) {
```

```
            return false;
        } else {
            return true;
        }
    }
```

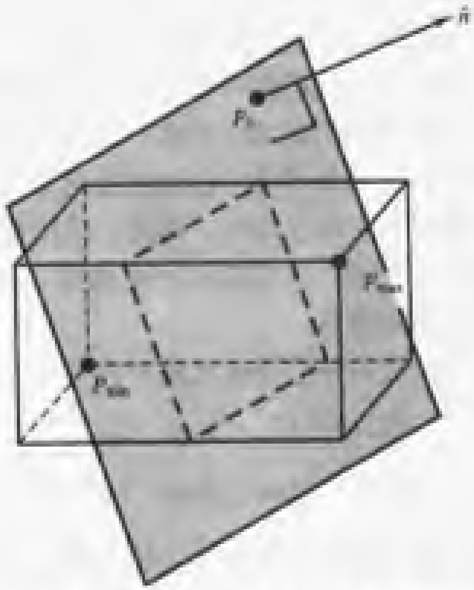


图 11.67　平面与轴对齐有界箱的相交

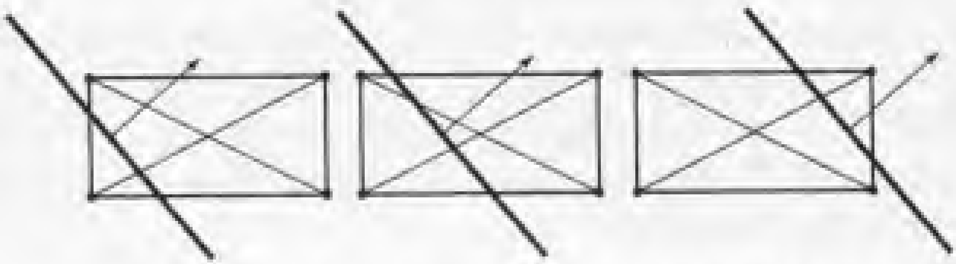


图 11.68　我们只需检查与平面法线最平齐的对角线端点所在的角

11.12.5　平面对象与有向有界箱的相交

我们在此考虑确定一个平面与一个有向有界箱是否相交的问题，基于上一节给出的原因，我们还是不计算确切的交点。图 11.69 说明了这个问题。

由于有向有界箱就是一个位于变换后的坐标系中的轴对齐有界箱，Möller 和 Haines（1999）认为，我们可以将平面法线变换到有向有界箱的中心和基底向量所定义的坐标系中，并应用我们用于处理轴对齐有界箱的算法。变换后的法线向量为

$$\hat{n}' = [\hat{u}\cdot\hat{n} \quad \hat{v}\cdot\hat{n} \quad \hat{w}\cdot\hat{n}]$$

这个变换后的法线便可用于平面一轴对齐有界箱相交算法中，而算法的其他部分都保持不变。然而，回想一下，轴对齐有界箱是用一对点来定义的，而有向有界箱是用一个位置、方向坐标系，以及半宽、半高和半深来定义的。因此，为了利用轴对齐有界箱算法，我们就必须在有向有界箱的中心和基底向量所定义的坐标系中找到有向有界箱的最小和最大点，然后再利用轴对齐有界箱算法。

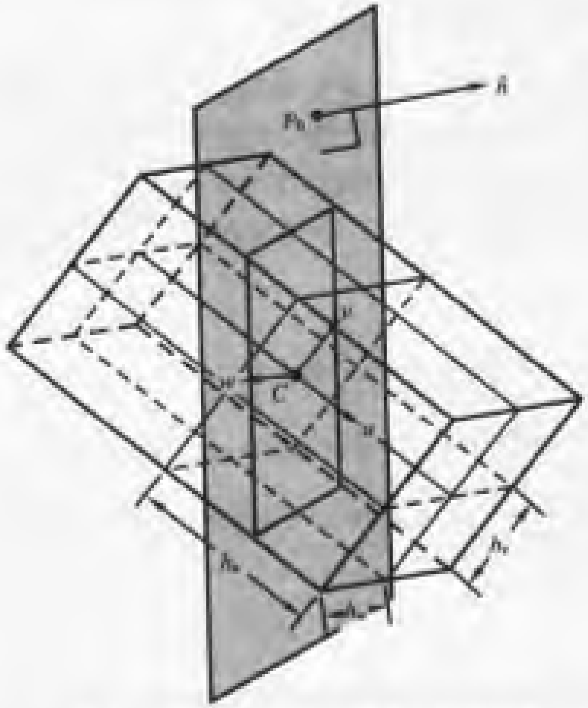


图 11.69 平面与有向有界箱之间的相交

Möller 和 Haines（1999）引用的另一种方法可能更好一些。我们将有向有界箱的对角线投影到平面的法线上，那么，如果该平面与任一条对角线的投影相交，则箱子与这个平面相交。这里采用的技巧是：不要按照上述的说明来实现，因为这样做可能得不到最好的效率。我们采用的方法是：如图 11.70 所示，我们不能简单地将缩放过的基底向量投影到该平面的法线上，相反，我们取每一个缩放过的基底向量的投影长度，并将它们相加。这样就得到了最长的投影对角线的半长：

$$d = \|h_u\hat{u} \cdot \hat{n}\| + \|h_v\hat{v} \cdot \hat{n}\|$$

显然，如果C与平面$\mathcal{P}$之间的（无符号）距离小于d，那么箱子与该平面相交，否则它们不相交。当然，可以对它们的平方值进行比较，以省去平方根计算。

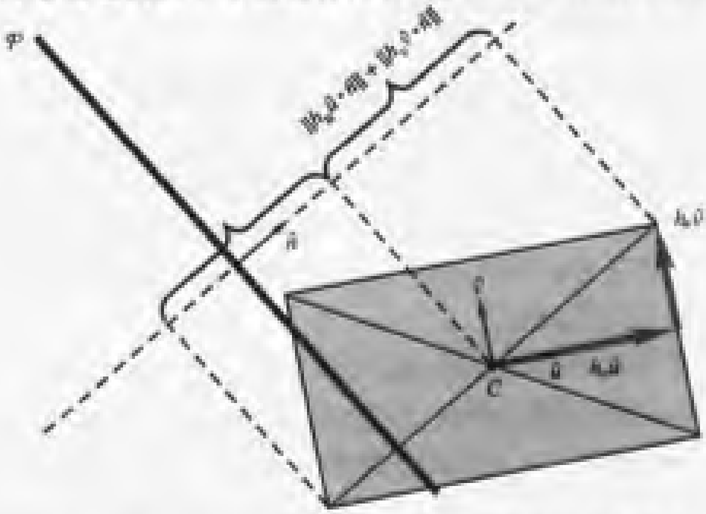


图 11.70 将一个 OBB 的对角线投影到平面的法线上

11.12.6　轴对齐有界箱之间的相交

利用轴对齐有界箱的面都垂直于其坐标系的基底向量这一性质，可以简化轴对齐有界箱之间的相交问题。图 11.71 显示了两个相交的轴对齐有界箱。这里利用的技巧是，在每一个基底向量方向上进行一次不重叠的轴对齐有界箱检测：如果轴对齐有界箱在任何一个方向上都不重叠，那么它们必定不相交；如果轴对齐有界箱在所有方向上都重叠，那么它们一定相交。

伪码为

```
boolean AABBIntersectAABB(AABB a, AABB b)
{
    // Check if AABBs fail to overlap in any direction
    foreach (dir in {x, y, z}) {
        if (a.min[dir] > b.max[dir] || b.min[dir] > a.max[dir]) {
            return false;
        }
    }

    // AABBs overlapped in all directions, so they intersect
    return true;
}
```

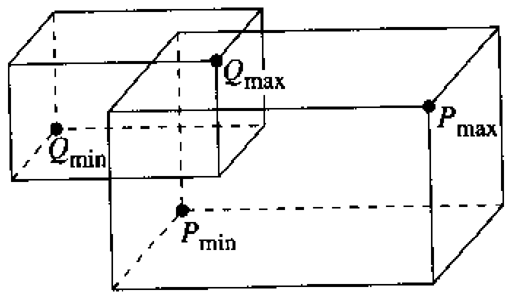


图 11.71　两个轴对齐有界箱之间的相交

11.12.7　有向有界箱之间的相交

在本节中，我们讨论检测两个有向有界箱之间的相交问题。在 11.12.3 节中，一个有向有界箱是用一个中心C，一个符合右手法则的正交基底$\{\hat{u}, \hat{v}, \hat{w}\}$和一个半长组$\{h_{\hat{u}}, h_{\hat{v}}, h_{\hat{w}}\}$来定义的。由于有向有界箱用来包围其他的图元，通过筛选掉不可能相交的情形，以达到提高相交、选取或者（可能的）着色等操作的目的，因此我们仅需考虑计算是否存在相交。如果有向有界箱相交，那么它们包围的图元并不一定相交，因此我们必须进行对它们所包围的对象进行针对对象的相交检测。

我们这里介绍的算法来自于 Gottschalk，Lin 和 Manocha（1996）。采用这种算法的原因是：最基本的方法就是简单地检测每一个有向有界箱的每一条边是否与另一个有向有界箱的每一个面相交，需要进行 144 次的边－面相交测试。更加有效的方法是利用分离轴检测，这种方法以如下的定理为基础：任何两个不重叠的多面体总是可以用一个既不与任何一个多面体的面平行也不与任何一个多面体的边平行的平面来分开。在二维空间中的说明

将帮助我们更清楚地理解这一点，如图 11.72 所示。分离有向有界箱A和B的平面（二维空间中的直线）在图中显示为点线。

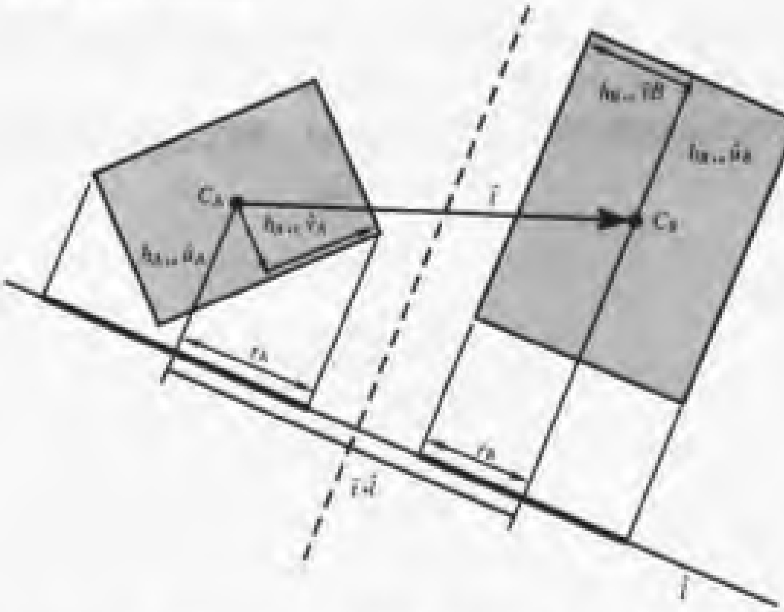


图 11.72　OBB 相交检测的二维示意图（Gottschalk, Lin 和 Manocha 1996）

每一个有向有界箱都具有 3 个面方向和 3 个边方向。这样我们就需要检测 15 个轴——取自每一个箱子的 3 个面和 9 种边的组合。如果有向有界箱不重叠，那么至少存在一个分离轴；如果有向有界箱重叠，则不存在分离轴。注意，一般地，如果有向有界箱不重叠，那么可以在少于 15 次的检测中找到一个分离轴。

基本的检测如下：

（1）选择一个要检测的轴。

（2）将有向有界箱的中心投影到该轴上。

（3）计算区间r_A和r_B的半径。

（4）如果半径之和小于两个有向有界箱的中心C_A和C_B在选定的轴上的投影之间的距离，那么区间是不重叠的，并且有向有界箱也是不重叠的。

这里采用的提高效率的“技巧”是将中心C_A和方向$\{\hat{u}_A, \hat{v}_A, \hat{w}_A\}$分别作为一个（坐标）坐标系原点和基底。然后，将有向有界箱B看成是由一个相对于A的旋转和平移变换T所得到的。这样，**R** 的三个列刚好就是三个向量$\{\hat{u}_B, \hat{v}_B, \hat{w}_B\}$。

通过用与它们的相关的半维（宽、高、深）来缩放有向有界箱的基底，将每一个基底投影到分离轴上，并相加，就可以得到投影的半径r_A和r_B：

$$r_A = \sum_{i \in \{u,v,w\}} h_{A,i} |a_{A,i} \cdot \hat{l}|$$

其中$a_{A,i}$是分别与$\hat{u}$, $\hat{v}$或$\hat{w}$相关的A的轴，对r_B也做类似的处理。

如果我们设$\vec{t} = C_B - C_A$，那么如果满足如下条件，则该区间不重叠：

$$|\vec{t} \cdot \hat{l}| > r_A + r_B$$

需要考虑 3 种基本的情形：检测的轴平行于A的一条边、平行于B的一条边、或者是

分别取自 A 和 B 的一对边的组合。

1. $\hat{l}$ 平行于 A 的一条边

这是最简单的情形。因为我们使用 C_A 和 $\{\hat{u}_A, \hat{v}_A, \hat{w}_A,\}$ 作为一个基底，所以轴 $\{\hat{a}_{A,u}, \hat{a}_{A,v}, \hat{a}_{A,w}\}$ 分别为 $[1\ \ 0\ \ 0]$，$[0\ \ 1\ \ 0]$ 和 $[0\ \ 0\ \ 1]$。

例如，如果我们正在检测平行于 $\hat{u}_A$ 的轴，那么 C_A 和 C_B 之间的投影距离为

$$
\begin{aligned}
|\vec{t}\cdot\hat{l}| &= |\vec{t}\cdot a_{A,u}| \\
&= |x_{\vec{t}}|
\end{aligned}
$$

A 的投影半径为

$$r_A = \sum_{i\in\{u,v,w\}} h_{A,i}|a_{A,i}\cdot\hat{l}|$$

然而，由于

$$\hat{l} = \hat{u}_A$$

我们有

$$
\begin{aligned}
r_A &= \sum_{i\in\{u,v,w\}} h_{A,i}|a_{A,i}\cdot a_{A,u}| \\
&= h_{A,u}
\end{aligned}
$$

B 的投影半径为

$$r_B = \sum_{i\in\{u,v,w\}} h_{B,i}|a_{B,i}\cdot\hat{l}|$$

然而，由于

$$\hat{l} = \hat{u}_a$$

我们有

$$
\begin{aligned}
r_B &= \sum_{i\in\{u,v,w\}} h_{B,i}|a_{B,i}\cdot a_u| \\
&= h_{B,u}|\mathrm{R}_{00}| + h_{B,v}|\mathrm{R}_{01}| + h_{B,w}|\mathrm{R}_{02}|
\end{aligned}
$$

$\hat{l} = \hat{v}_A$ 和 $\hat{l} = \hat{w}_A$ 的情形与此相似。

2. $\hat{l}$ 平行于 B 的一条边

这种情形与 $\hat{l}$ 平行于 A 的一条边的情形几乎同样简单。例如，如果我们正在检测平行于 $\hat{u}_B$ 的轴，那么 C_A 和 C_B 之间的投影距离为

$$
\begin{aligned}
|\vec{t}\cdot\hat{l}| &= |\vec{t}\cdot a_{B,u}| \\
&= |t_x\mathrm{R}_{00} + t_y\mathrm{R}_{10} + t_z\mathrm{R}_{20}|
\end{aligned}
$$

A 的投影半径为

$$r_A = \sum_{i\in\{u,v,w\}} h_{A,i}|a_{B,i}\cdot\hat{l}|$$

然而，由于

$$\hat{l}=\hat{u}_B$$

我们有

$$\begin{aligned} r_A &= \sum_{i\in\{u,v,w\}} h_{A,i}|a_{A,i}\cdot a_{B,u}| \\ &= h_{A,u}|\mathbf{R}_{00}|+h_{A,v}|\mathbf{R}_{10}|+h_{A,w}|\mathbf{R}_{20}| \end{aligned}$$

B的投影半径为

$$r_B=\sum_{i\in\{u,v,w\}} h_{B,i}|a_{B,i}\cdot\hat{l}|$$

然而，由于

$$\hat{l}=\hat{u}_a$$

我们有

$$\begin{aligned} r_B &= \sum_{i\in\{u,v,w\}} h_{B,i}|a_{B,i}\cdot a_u| \\ &= h_{B,u} \end{aligned}$$

$\hat{l}=\hat{v}_B$ 和$\hat{l}=\hat{w}_B$的情形与此相似。

3．$\hat{l}$是分别取自A和B的一对边的组合

对于检测的轴是分别取自A和B的一对边的组合的情形，我们使用A和B的基底向量的叉积向量。

例如，如果我们正在检测平行于$\hat{u}_A\times\hat{v}_B$的轴，那么C_A 和C_B 之间的投影距离为

$$\begin{aligned} |\vec{t}\cdot\hat{l}| &= |\vec{t}\cdot(a_{A,u}\times a_{B,v})| \\ &= |\vec{t}\cdot[\,0,-a_{B,v,z},a_{B,v,y}\,]| \\ &= |t_z\mathbf{R}_{11}-t_y\mathbf{R}_{21}| \end{aligned}$$

A的投影半径为

$$r_A=\sum_{i\in\{u,v,w\}} h_{A,i}|a_{A,i}\cdot\hat{l}|$$

然而，由于

$$\hat{l}=(a_{A,u}\times a_{B,v})$$

我们有

$$\begin{aligned} r_A &= \sum_{i\in\{u,v,w\}} h_{A,i}|a_{A,i}\cdot(a_{A,u}\times a_{B,v})| \\ &= \sum_{i\in\{u,v,w\}} h_{A,i}|a_{B,v}\cdot(a_{A,u}\times a_{A,i})| \\ &= h_{A,v}|\mathbf{R}_{21}|+h_{A,w}|\mathbf{R}_{11}| \end{aligned}$$

B 的投影半径为

$$r_B = \sum_{i \in \{u,v,w\}} h_{B,i} |a_{B,i} \cdot \hat{l}|$$

然而，由于

$$\hat{l} = (a_{A,u} \times a_{B,v})$$

我们有

$$\begin{aligned} r_B &= \sum_{i \in \{u,v,w\}} h_{B,i} |a_{B,i} \cdot (a_{A,u} \times a_{B,v})| \\ &= \sum_{i \in \{u,v,w\}} h_{B,i} |a_{A,v} \cdot (a_{B,i} \times a_{B,v})| \\ &= h_{B,u} |R_{02}| + h_{B,w} |R_{00}| \end{aligned}$$

11.12.8 球面与轴对齐有界箱的相交

在本节中，我们讨论球面与轴对齐有界箱的相交问题。用相对的两个具有最小和最大分量的两个角来定义轴对齐有界箱

$$P_{\min} = [\, x_{\min} \quad y_{\min} \quad z_{\min} \,]$$

$$P_{\max} = [\, x_{\max} \quad y_{\max} \quad z_{\max} \,]$$

而球面可以简单地用其球心 C 和半径 r 来定义，如图 11.73 所示。

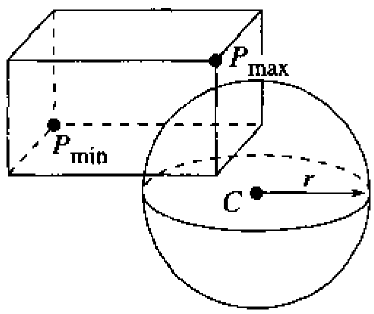


图 11.73　轴对齐有界箱与球面的相交

Arvo（1990）提供的一种算法描述了一种确定球面与轴对齐有界箱是否相交的一种非常聪明的方法。其基本思想是，在轴对齐有界箱上（或内部）找到一个与球面的球心最接近的点：如果其距离平方小于球面的半径平方，那么它们相交；否则，它们不相交。其聪明之处在于，Arvo 发现了找到这个最接近的点的有效方法。轴对齐有界箱上（或内部）的最接近的点使如下的距离平方函数最小化

$$\mathrm{dist}^2(Q) = (C_x - Q_x)^2 + (C_y - Q_y)^2 + (C_z - Q_z)^2$$

满足

$$P_{\min,x} \leqslant Q_x \leqslant Q_x$$

$$P_{\min,y} \leqslant Q_y \leqslant Q_y$$

$$P_{\min,z} \leqslant Q_z \leqslant Q_z$$

Arvo 注意到，我们能分别找到最小距离点的分量，然后组合这些分量，就得到了最小距离点。

伪码为

```
boolean AABBIntersectSphere(AABB box, Sphere sphere)
{
    // Initial distance is 0
    float distSquared = 0;

    // Compute distance in each direction,
    // summing as we go.
    foreach (dir in {x, y, z}) {
        if (sphere.center[dir] < box.min[dir]) {
            distSquared += Square((sphere.c[dir] - box.min[dir]));
        } else if (sphere.center[dir] > box.max[dir]) {
            distSquared += Square((sphere.c[dir] - box.max[dir]));
        }
    }

    // Compare distance to radius squared
    if (distSquared <= sphere.radius * sphere.radius) {
        return true;
    } else {
        return false;
    }
}
```

Arvo 还提供了一种处理“空的”图元的一种变体方法，在这种情形中，其中一个完全包含另一个就意味着“不相交”；Arvo 还提供了一种将上述方法扩展到椭球面（然而，应该注意，这种方法仅适用于轴平行于其定义坐标系的基底向量的椭球面）变体方法。

11.12.9 圆柱面之间的相交

本节介绍如何确定两个封闭的圆柱面是否相交。这里使用的算法其方法与 11.11 节讨论的分离轴方法相同，尽管这里的问题比我们在处理凸多面体时遇到的问题更加复杂，因为封闭的圆柱面并不是多面体。如果想在实时的图形引擎中用圆柱面来计算包围的体积，那么最后得出的算法是一种相当费时的算法。比圆柱面更好的一种选择是囊形，即与一条线段等距的点集。当且仅当两个囊的线段之间的距离小于或等于囊的半径之和时，两个囊才相交，这种检测能节省很多时间。

除了使用依赖于投影到直线上的分离轴方法外，这种算法还要求通过投影到平面上而分离。这个概念类似于分离轴。如果存在一个平面，两个凸对象投影到该平面上所得的区域不相交，那么这两个对象不相交。与使用直线的情形完全一样，考虑包含原点的平面就足够了。给定一个包含原点并具有单位长度法线$\hat{n}$的平面，凸集C在该平面上的投影为如下的点集

$$R = \{Y : Y = X - (\hat{n}\cdot X)\hat{n} = (\mathbf{I} - \hat{n}\hat{n}^{\mathrm{T}})X, \quad X \in C\}$$

其中 $\mathbf{I}$ 是 3×3 的单位矩阵。投影集本身也是一个凸集。如果存在一个法线$\hat{n}$使得两个凸集C_0和C_1的投影集R_0和R_1不相交，即$R_0 \cap R_1 = \emptyset$，那么两个凸集是分离的。这个条件的确定可能涉及一种或多种几何方法，例如，需要证明两个集合之间的距离是正的。也可以分析在本地二维空间上的投影，并试图找到在二维空间上的一条分离直线，而且这种方法应该也可以用于三维空间。

1. 圆柱面的表示方法

一个圆柱面具有一个中心点C，单位长度轴方向$\hat{w}$，半径r和高度h。圆柱面的端圆盘位于$C \pm (h/2)\hat{w}$。设$\hat{u}$和$\hat{v}$是单位长度向量，从而$\{\hat{u}, \hat{v}, \hat{w}\}$是一个正交向量的右手集。即这些向量是单位长度、互相正交的，并且$\hat{w} = \hat{u} \times \hat{v}$。位于圆柱曲面上的点可以用参数形式表示为

$$X(\theta, t) = C + (r\cos\theta)\hat{u} + (r\sin\theta)\hat{v} + t\hat{w}, \quad \theta \in [0, 2\pi),\ |t| \leqslant h/2$$

端圆盘可，用参数形式表示为

$$X(\theta, \rho) = C + (\rho\cos\theta)\hat{u} + (\rho\sin\theta)\hat{v} \pm (h/2)\hat{w}, \quad \theta \in [0, 2\pi),\ \rho \in [0, r]$$

圆柱面在一条直线或一个平面上的投影完全由圆柱面的墙确定，而不由端圆盘确定，因此，第二个方程与相交检测无关。

$\hat{u}$和$\hat{v}$的选取是任意的。圆柱面之间的相交查询应该与这种选择无关，然而，对于特定的选择，有的算法可能更简单一些。表示圆柱面的墙的二次方程为$(X - C)^{\mathrm{T}}(\mathbf{I} - \hat{w}\hat{w}^{\mathrm{T}})(X - C) = r^2$。圆柱面的边界可以表示为$|\hat{w}\cdot(X - C)| \leqslant h/2$。这种表示法仅与$C, \hat{w}, r$和$h$有关。

2. 圆柱面在直线上的投影

设直线为$s\vec{d}$，其中$\vec{d}$是一个非零向量。一个圆柱面点在该直线上的投影为

$$\lambda(\theta, t) = \vec{d}\cdot X(\theta, t) = \vec{d}\cdot C + (r\cos\theta)\vec{d}\cdot\hat{u} + (r\sin\theta)\vec{d}\cdot\hat{v} + t\vec{d}\cdot\hat{w}$$

投影区间的端点由该式的极值点所确定。当所有与参数相关的 3 个项都设为尽可能大时，可得到最大值。t 项的最大值为$(h/2)|\vec{d}\cdot\hat{w}|$。$\theta$ 项，并不包括半径，可以看做一个点集$(\cos\theta, \sin\theta)\cdot(\vec{d}\cdot\hat{u}, \vec{d}\cdot\hat{v})$。当$(\cos\theta, \sin\theta)$与$(\vec{d}\cdot\hat{u}, \vec{d}\cdot\hat{v})$的方向相同时，该值就是最大值。因此，

$$(\cos\theta, \sin\theta) = \frac{(\vec{d}\cdot\hat{u}, \vec{d}\cdot\hat{v})}{\sqrt{(\vec{d}\cdot\hat{u})^2 + (\vec{d}\cdot\hat{v})^2}}$$

而且最大投影值为

$$\lambda_{\max} = \vec{d}\cdot C + r\sqrt{\|\vec{d}\|^2 - (\vec{d}\cdot\hat{w})^2} + (h/2)|\vec{d}\cdot\hat{w}|$$

其中，我们利用了如下的事实：$\vec{d} = (\vec{d}\cdot\hat{u})\hat{u} + (\vec{d}\cdot\hat{v})\hat{v} + (\vec{d}\cdot\hat{w})\hat{w}$，其中隐含了$(\vec{d}\cdot\hat{u})^2 + (\vec{d}\cdot\hat{v})^2 + (\vec{d}\cdot\hat{w})^2 = \|\vec{d}\|^2$。最小投影值可，用类似的方法导出：

$$\lambda_{\min} = \vec{d}\cdot C - r\sqrt{\|\vec{d}\|^2 - (\vec{d}\cdot\hat{w})^2} - (h/2)|\vec{d}\cdot\hat{w}|$$

3. 圆柱面在平面上的投影

设平面为$\hat{n}\cdot X = 0$，其中$\hat{n}$是一个单位长度法线。圆柱面在平面上的投影为如下 3 种几

何构形之一：

（1）当$\hat{w}$平行于$\hat{n}$时为一个圆盘。

（2）当$\hat{w}$垂直于$\hat{n}$时为一个矩形。

（3）一个圆角矩形。

投影矩阵是$\mathrm{P} = \mathrm{I} - \hat{n}\hat{n}^{\mathrm{T}}$。在第一种情形中，圆盘的中心是$\mathrm{P}C$，半径为$r$。在第二种情形中，矩形的中心是$\mathrm{P}C$，单位长度轴方向为$\hat{w}$和$\hat{w} \times \hat{n}$。矩形的 4 个角为$\mathrm{P}C \pm r\hat{w} \times \hat{n} \pm (h/2)\hat{w}$。

第三种情形也只是稍微复杂一些。投影区域的中心为$\mathrm{P}C$。投影方向的轴为非单位长度向量$\mathrm{P}\hat{w}$。圆柱面的一个位于平面上并垂直于$\hat{n}$的轴的方向为$\hat{u} = (\mathrm{P}\hat{w}) \times \hat{n}/\|(\mathrm{P}\hat{w}) \times \hat{n}\|$。映射为投影矩形区域的 4 个角的圆柱面上的 4 个点为$C \pm r\hat{u} \pm (h/2)\hat{w}$。4 个角为$\mathrm{P}C \pm r\hat{u} \pm (h/2)\mathrm{P}\hat{w}$。

设$\hat{v} = \hat{w} \times \hat{u}$。圆柱面的端圆为$X(\theta) = C \pm r((\cos\theta)\hat{u} + (\sin\theta)\hat{v}) \pm (h/2)\hat{w}$。设$Y = \mathrm{P}(X - C \pm (h/2)\hat{w})$，则$Y = r((\cos\theta)\hat{u} + (\sin\theta)\mathrm{P}\hat{v})$。因此，$\hat{u} \cdot Y = r\cos\theta$和$\mathrm{P}\hat{v} \cdot Y = \|\mathrm{P}\hat{v}\|^2 r\sin\theta$。将它们组合在一起，可得

$$
\begin{aligned}
1 &= \frac{1}{r^2}\left((\hat{u} \cdot Y)^2 + \frac{1}{\|\mathrm{P}\hat{v}\|^4}(\mathrm{P}\hat{v} \cdot Y)^2\right) \\
&= \frac{1}{r^2}Y^{\mathrm{T}}\left(\hat{u}\hat{u}^{\mathrm{T}} + \frac{1}{\|\mathrm{P}\hat{v}\|^2}\frac{\mathrm{P}\hat{v}}{\|\mathrm{P}\hat{v}\|}\frac{\mathrm{P}\hat{v}}{\|\mathrm{P}\hat{v}\|}^{\mathrm{T}}\right)Y \\
&= (\mathrm{P}(X - C \pm (h/2)\hat{w}))^{\mathrm{T}}\left(\frac{1}{r^2}\hat{u}\hat{u}^{\mathrm{T}} + \frac{1}{r^2\|\mathrm{P}\hat{v}\|^2}\frac{\mathrm{P}\hat{v}}{\|\mathrm{P}\hat{v}\|}\frac{\mathrm{P}\hat{v}}{\|\mathrm{P}\hat{v}\|}^{\mathrm{T}}\right)(\mathrm{P}(X - C \pm (h/2)\hat{w}))
\end{aligned}
$$

这是表示两个椭圆的方程，它们的中心为$\mathrm{P}(C \pm (h/2)\hat{w})$，轴为$\hat{u}$和$\mathrm{P}\hat{v}/\|\mathrm{P}\hat{v}\|$，轴半长为$r$和$r\|\mathrm{P}\hat{v}\|$。

4. 两个圆柱面的分离直线检测

给定两个圆柱面，其中心为C_i，轴方向为$\hat{w}_i$，半径为r_i，高度为h_i，其中$i = 0，1$。这两个圆柱面是分离的，如果存在一个非零方向$\vec{d}$使得

$$\vec{d} \cdot C_0 - r_0\sqrt{\|\vec{d}\|^2 - (\vec{d} \cdot \hat{w}_0)^2} - (h_0/2)|\vec{d} \cdot \hat{w}_0| > \vec{d} \cdot C_1 + r_1\sqrt{\|\vec{d}\|^2 - (\vec{d} \cdot \hat{w}_1)^2} + (h_1/2)|\vec{d} \cdot \hat{w}_1|$$

或者

$$\vec{d} \cdot C_0 + r_0\sqrt{\|\vec{d}\|^2 - (\vec{d} \cdot \hat{w}_0)^2} + (h_0/2)|\vec{d} \cdot \hat{w}_0| < \vec{d} \cdot C_1 - r_1\sqrt{\|\vec{d}\|^2 - (\vec{d} \cdot \hat{w}_1)^2} - (h_1/2)|\vec{d} \cdot \hat{w}_1|$$

之一成立。定义$\vec{\Delta} = C_1 - C_0$，上述检测可改写为一个表达式，即$f(\vec{d}) < 0$，其中

$$f(\vec{d}) = r_0\|\mathrm{P}_0\vec{d}\| + r_1\|\mathrm{P}_1\vec{d}\| + (h_0/2)|\vec{d} \cdot \hat{w}_0| + (h_1/2)|\vec{d} \cdot \hat{w}_1| - |\vec{d} \cdot \vec{\Delta}|$$

并且$\mathrm{P}_i = \mathrm{I} - \hat{w}_i\hat{w}_i^{\mathrm{T}}$，其中$i = 0，1$。

如果$\vec{\Delta} = 0$，则$f \geqslant 0$。这在几何上是显而易见的，因为两个具有相同中心的圆柱面已经相交。其余的讨论假设$\vec{\Delta} \neq \vec{0}$。如果$\vec{d}$垂直于$\vec{\Delta}$，那么$f(\vec{d}) \geqslant 0$。这说明任意垂直于包含两个圆柱面中心直线的直线不可能是分离轴。这在几何上也是显而易见的。直线$C_0 + s\vec{\Delta}$与两个圆柱面都相交于它们的中心。如果将两个圆柱面投影到平面$\vec{\Delta} \cdot (X - C_0) = 0$上，那么

两个投影区域重叠。选择平面上任何包含C_0的直线，该直线都与两个投影区域相交。

如果$\vec{d}$是分离方向，则$f(\vec{d})<0$。由于$f(t\vec{d})=|t|f(\vec{d})$，因此对任何$t$有$f(t\vec{d})<0$。这与几何意义是一致的。分离方向的任何非零乘积本身也必定是一个分离方向。这样，我们就能专注于单位球面，即$|\vec{d}|=1$。函数f在单位球面上（即一个紧凑集上）是连续的，因此，f必定会在该球面上的某点取得其最小值。这是一个二维空间上的最小值问题，然而，球面几何使分析变得复杂了一些。对可能的分离方向集合的不同限制条件可能使最小值问题出现在直线或者平面而不是在球面上。

对f的分析涉及计算其导数$\vec{\nabla}(f)$和确定其临界点。这些点满足$\vec{\nabla}(f)$为零或者无定义。很容易指定后一种类型。当5个绝对值符号内的任意项为零时，梯度无定义。因此，$\vec{\nabla}(f)$在$\hat{w}_0, \hat{w}_1$处、垂直于$\hat{w}_0$的向量处、垂直于$\hat{w}_1$的向量处、垂直于$\vec{\Delta}$的向量处都无定义。我们已经证明了$f\geqslant 0$将使向量垂直于$\vec{\Delta}$，因此，我们可以忽略这种情形。

5. 在$\hat{w}_0, \hat{w}_1$和$\hat{w}_0\times\hat{w}_1$处的检测

可以检测圆柱面的轴方向本身是否为分离方向。检测函数值为

$$f(\hat{w}_0)=r_1\|\hat{w}_0\times\hat{w}_1\|+(h_0/2)+(h_1/2)|\hat{w}_0\cdot\hat{w}_1|-|\hat{w}_0\cdot\vec{\Delta}|$$

和

$$f(\hat{w}_1)=r_0\|\hat{w}_0\times\hat{w}_1\|+(h_0/2)|\hat{w}_0\cdot\hat{w}_1|+(h_1/2)-|\hat{w}_1\cdot\vec{\Delta}|$$

如果其中有一个函数的值为负数，则两个圆柱面相交。可以避免其中的平方根运算。例如，$f(\hat{w}_0)<0$等价于

$$r_1\|\hat{w}_0\times\hat{w}_1\|<|\hat{w}_0\cdot\vec{\Delta}|-h_0/2-(h_1/2)|\hat{w}_0\cdot\hat{w}_1|=:\rho$$

判断右式。如果$\rho\leqslant 0$，则不等式不可能为真，因为$\hat{w}_0\times\hat{w}_1\neq\vec{0}$和左式都为正。否则，$\rho>0$，并且可以判断出$r_1\|\hat{w}_0\times\hat{w}_1\|^2<\rho^2$。相似的方法可应用于$f(\hat{w}_1)<0$的情形。

最后的一个检测不需要进行多少运算，并且可能得出一个快速的不相交判断条件，即

$$f(\hat{w}_0\times\hat{w}_1)=(r_0+r_1)\|\hat{w}_0\times\hat{w}_1\|-|\hat{w}_0\times\hat{w}_1\cdot\vec{\Delta}|<0$$

或者其等价式

$$(r_0+r_1)^2\|\hat{w}_0\times\hat{w}_1\|^2<|\hat{w}_0\times\hat{w}_1\cdot\vec{\Delta}|^2$$

当然，其中假设$\hat{w}_0\times\hat{w}_1\neq\vec{0}$。该向量精确地说明了$f$的梯度在什么情形下是没有定义的。

如果$\hat{w}_0$和$\hat{w}_1$是平行的，那么$\hat{w}_0\times\hat{w}_1=\vec{0}$并且$|\hat{w}_0\cdot\hat{w}_1|=1$。恒等式

$$(\vec{a}\times\vec{b})\cdot(\vec{c}\times\vec{d})=(\vec{a}\cdot\vec{c})(\vec{b}\cdot\vec{d})-(\vec{a}\cdot\vec{d})(\vec{b}\cdot\vec{c})$$

可用来证明$\|\hat{w}_0\times\hat{w}_1\|^2=1-(\hat{w}_0\cdot\hat{w}_1)^2$。$\hat{w}_0$的检测函数就是

$$f(\hat{w}_0)=(h_0+h_1)/2-|\hat{w}_0\cdot\vec{\Delta}|$$

如果$(h_0+h_1)/2<|\hat{w}_0\cdot\vec{\Delta}|$，则两个圆柱面是分离的。如果$f(\hat{w}_0)\geqslant 0$，则两个圆柱面可能由一个垂直于$\hat{w}_0$的方向所分离。在几何上可以确定两个圆柱面在平面$\hat{w}_0\cdot X=\vec{0}$上投影所得到的圆是否相交。当且仅当$\vec{\Delta}$在平面上的投影长度大于两个圆的半径之和时，这两个圆不相交。$\vec{\Delta}$的投影是$\vec{\Delta}-(\hat{w}_0\cdot\vec{\Delta})\hat{w}_0$，其平方长度为

$$\|\vec{\Delta}-(\hat{w}_0\cdot\vec{\Delta})\hat{w}_0\|^2=\|\vec{\Delta}\|^2-(\hat{w}_0\cdot\vec{\Delta})^2$$

这两个圆的半径之和就是两个圆柱面的半径之和，即r_0+r_1。因此，如果$\|\vec{\Delta}\|^2-(\hat{w}_0\cdot\vec{\Delta})^2>(r_0+r_1)^2$，则两个圆柱面是分离的。

在本节的其余部分中，我们假定$\hat{w}_0$和$\hat{w}_1$是不平行的。

6. 在垂直于$\hat{w}_0$或$\hat{w}_1$的向量处的检测

设f的定义域为一个单位球面，垂直于$\hat{w}_0$的向量集合是该球面的一个大圆。f的梯度在这个大圆上无定义。定义$\vec{d}(\theta)=(\cos\theta)\hat{u}_0+(\sin\theta)\vec{v}_0$和$F(\theta)=f(\vec{d}(\theta))$。如果我们能证明$F(\theta)<0$对于某些$\theta\in[0,2\pi)$成立，那么对应的方向就是这两个圆柱面的分离线。然而，F却是一个复杂的函数，并不能用简单的方法来分析它。由于$f(-\vec{d})=f(\vec{d})$，因此我们可以专注于分析半个大圆。我们并不将f限制为半个圆，而是限制于一条切线$\vec{d}(x)=x\hat{u}_0+\hat{v}_0$，并定义$F(x)=f(\vec{d}(x))$，因此

$$\begin{aligned}F(x)&=r_0\sqrt{x^2+1}+r_1\|(\mathrm{P}_1\hat{u}_0)x+(\mathrm{P}_1\hat{v}_0)\|+(h_1/2)|(\hat{w}_1\cdot\hat{u}_0)x\\&\quad+(\hat{w}_1\cdot\hat{v}_0)|-|(\vec{\Delta}\cdot\hat{u}_0)x+(\vec{\Delta}\cdot\hat{v}_0)|\\&=r_0\sqrt{x^2+1}+r_1\|\vec{a}_0x+\vec{b}_0\|+(h_1/2)\|\vec{a}_1x+\vec{b}_1\|-\|\vec{a}_2x+\vec{b}_2\|\end{aligned}$$

在上式中用符号标志来替换最后的两个绝对值，将其分解为 4 种情形，可以更方便地分析该函数：

$$G(x)=r_0\sqrt{x^2+1}+r_1\|\vec{a}_0x+\vec{b}_0\|+\sigma_1(h_1/2)(\vec{a}_1x+\vec{b}_1)-\sigma_2(\vec{a}_2x+\vec{b}_2)$$

其中$|\sigma_1|=|\sigma_2|=1$。通过计算$G'(x)$，并确定它在何处为零或无定义，就能对每一种(σ_1,σ_2)的选择计算G的最小值。必须首先检测任何的临界点x，以确定它是否与选择的符号一致。即必须检测临界点，以确定$\sigma_1(a_1x+b_1)\geqslant 0$和$\sigma_2(a_2x+b_2)\geqslant 0$。如果它们成立，那么计算$G(x)$，并将其与零比较。其导数为

$$G'(x)=r_0\frac{x}{\sqrt{x^2+1}}+r_1\vec{a}_0\cdot\frac{\vec{a}_0x+\vec{b}_0}{\|\vec{a}_0x+\vec{b}_0\|}+(\sigma_1h_1/2)\vec{a}_1-\sigma_2\vec{b}_2$$

当$\|\vec{a}_0x+\vec{b}_0\|=0$时，该导数无定义，但是，我们在前面讨论过，当原始方向平行于$\hat{w}_0\times\hat{w}_1$时，就会出现这种情形。利用代数方法求解$G'(x)=0$，可以应用几次平方运算。注意，$G'(x)=0$的形式为

$$L_0\sqrt{Q_0}+L_1\sqrt{Q_1}=c\sqrt{Q_0Q_1}$$

其中L_i是关于x的线性函数，Q_i是关于x的二次函数，而c是一个常数。平方并移项，可得

$$2L_0L_1\sqrt{Q_0Q_1}=c^2Q_0Q_1-L_0^2Q_0-L_1^2Q_1$$

再平方并移项，可得

$$4L_0^2L_1^2Q_0Q_1-(c^2Q_0Q_1-L_0^2Q_0-L_1^2Q_1)^2=0$$

左式是关于x的 8 次多项式。可用数值方法来求解其根，用前面介绍的方法来检测其有效性，然后可以检测G是否有效。

还有另一种方法。注意到，确定垂直于$\hat{w}_0$的分离方向等价于将这两个圆柱面投影到平面$\hat{w}_0 \cdot X = 0$上，并确定投影是否相交。第一个圆柱面投影为一个圆盘。第二个圆柱面可能投影为一个圆盘、一个矩形或者圆角矩形，这取决于$\hat{w}_1$与$\hat{w}_0$之间的关系。通过证明C_0的投影并不位于第二个圆柱面的投影之内，以及C_0与第二个圆柱面的投影之间的距离大于r_0，就能确定它们是分离的。如果第二个圆柱面的投影是一个圆盘，那么该距离就是$\vec{\Delta}$的投影的长度。如果第二个圆柱面的投影是一个矩形，那么问题变为计算在这个平面上的一个点与一个矩形之间的距离。这种检测并不费时。如果第二个圆柱面的投影是一个圆角矩形，那么问题变为计算一个点与一个矩形，以及该点与两个椭圆之间的距离的最小值，然后，将该值与r_0进行比较。计算在一个平面上的一个点与一个椭圆之间的距离需要计算一个 4 次多项式的根。在最坏的情形中，需要权衡选择求解一个 8 次多项式的根还是选择计算两个 4 次多项式的根。

7. $\vec{\nabla}(f) = \vec{0}$的方向检测

对称式$f(-\vec{d}) = f(\vec{d})$说明我们仅需在半个球上分析f；可以自动地确定在另外半个球上的值。由于在垂直于$\vec{\Delta}$的向量的大圆上有$f \geqslant 0$，我们可以专注于极点为$\hat{w} = \vec{\Delta}/\|\vec{\Delta}\|$的半球。我们可以投影到该极点的切面上，而不是投影到这个半球上。这个映射为$\vec{d} = x\hat{u} + y\hat{v} + \hat{w}$，其中$\hat{u}$，$\hat{v}$和$\hat{w}$构成一个右手正交系。定义旋转矩阵$\mathrm{R} = [\hat{u}|\hat{v}|\hat{w}]$和$\vec{\xi} = (x, y, 1)$，可将函数$f$简化为

$$F(x, y) = r_0\|\mathrm{P}_0\mathrm{R}\vec{\xi}\| + r_1\|\mathrm{P}_1\mathrm{R}\vec{\xi}\| + (h_0/2)|\hat{w}_0 \cdot \mathrm{R}\vec{\xi}| + (h_1/2)|\hat{w}_1 \cdot \mathrm{R}\vec{\xi}| - \|\vec{\Delta}\|$$

其中$(x, y) \in \mathbb{R}^2$。为了确定$F(x, y) < 0$对某些(x, y)成立，只需证明F的最小值是负数。当F的梯度为零或者无定义时，就可得到取得最小值的点。在开始的 4 个绝对值项之一为零的点上，$\vec{\nabla}(F)$无定义。在$\vec{d}$位于单位球面的点上，第一项在$\hat{w}_0$上为零，第二项在$\hat{w}_1$上为零，第三项在任何垂直于$\hat{w}_0$的向量上为零，第四项在任何垂直于$\hat{w}_1$的向量上为零。在检测了所有这样的点之后，就可以确定$F \geqslant 0$是否成立，检测分离的下一步就是计算$\vec{\nabla}(F) = \vec{0}$的解，并检测这些解是否满足$F < 0$。

如果$\vec{d}$是一个分离方向，那么$f(\vec{d}) < 0$。注意$f(t\vec{d}) = |t|f(\vec{d})$，因此，对任意的$t$有$f(t\vec{d}) < 0$。这与该问题的几何分析一致。分离方向与任意非零的数相乘得到的向量也必定是一个分离方向。这样，我们就可以专注于单位球面，即$|\vec{d}| = 1$。函数f在单位球面上是连续的，这是一个紧凑集，因此，f必定在球面上的某点处取得最小值。这是一个二维空间上的最小值问题，然而，球面几何使问题变得复杂化。可以做另外一种关于可能的分离方向的集合限制，这样将得到一个在平面上而不是在球面上的二维空间最小值问题。

首先，说明一下表示方法。函数$f(\vec{d})$可写为

$$f(\vec{d}) = r_0\|\mathrm{A}_0^{\mathrm{T}}\vec{d}\| + r_1\|\mathrm{A}_1^{\mathrm{T}}\vec{d}\| + (h_0/2)|\vec{d} \cdot \hat{w}_0| + (h_1/2)|\vec{d} \cdot \hat{w}_1| - |\vec{d} \cdot \vec{\Delta}|$$

其中矩阵$\mathrm{A}_i = [\hat{u}_i|\hat{v}_i]$是 3×2 矩阵。注意，$\mathrm{A}_i^{\mathrm{T}}\mathrm{A}_i = I_2$，2×2 的单位矩阵，以及$\mathrm{A}_i\mathrm{A}_i^{\mathrm{T}} = \mathrm{I}_3 - \hat{w}_i\hat{w}_i^{\mathrm{T}}$，其中$\mathrm{I}_3$是 3×3 的单位矩阵。

对称式$f(-\vec{d}) = f(\vec{d})$说明我们仅需在半个球上分析f；可以自动地确定在另外半个球上的值。在直接分析f时的复杂因子，其实就是绝对值项$|\vec{d} \cdot \hat{w}_0|$，$|\vec{d} \cdot \hat{w}_1|$和$|\vec{d} \cdot \vec{\Delta}|$。我们观察去掉了绝对值的函数。为了说明，设

$$g_0(\vec{d}) = r_0\|\mathbf{A}_0^{\mathrm{T}}\vec{d}\| + r_1\|\mathbf{A}_1^{\mathrm{T}}\vec{d}\| - \vec{d}\cdot\vec{\phi}$$

其中 $\vec{\phi} = \vec{\Delta} - (h_0/2)\hat{w}_0 - (h_1/2)\hat{w}_1$。如果分析 g_0 得到方向 $\vec{d}$ 满足 $g_0(\vec{d}) < 0$，并且如果 $\vec{d}\cdot\hat{w}_0 \geqslant 0, \vec{d}\cdot\hat{w}_1 \geqslant 0$ 和 $\vec{d}\cdot\vec{\Delta} \geqslant 0$，那么 $f(\vec{d}) < 0$，我们就得到一个分离方向。然而，即使在 $g_0(\vec{d}) < 0$ 时（在这种情形中，$\vec{d}$ 是分离线的可能被排除），这个不等式限制条件也可能不满足。相伴的函数是

$$g_1(\vec{d}) = r_0\|\mathbf{A}_0^{\mathrm{T}}\vec{d}\| + r_1\|\mathbf{A}_1^{\mathrm{T}}\vec{d}\| + \vec{d}\cdot\vec{\phi}$$

如果分析 g_1 得到方向 $\vec{d}$ 满足 $g_1(\vec{d}) < 0$，并且如果 $\vec{d}\cdot\hat{w}_0 \leqslant 0, \vec{d}\cdot\hat{w}_1 \leqslant 0$ 且 $\vec{d}\cdot\vec{\Delta} \leqslant 0$，那么 $f(\vec{d}) < 0$，我们就得到一个分离方向。然而，即使在 $g_1(\vec{d}) < 0$ 时（在这种情形中，$\vec{d}$ 是分离线的可能被排除），这个不等式限制条件也可能不满足。有 4 种这样的函数对需要考虑，穷举了 3 个绝对值项的所有 8 种可能的符号。

我们现在分析 $g_0(\vec{d})$。如果 $\vec{\phi} = \vec{0}$，那么很清楚，对所有方向都有 $g_0(\vec{d}) \geqslant 0$，因此不存在分离。在下面的讨论中，假定 $\vec{\phi} \neq \vec{0}$。任何满足 $\vec{d}\cdot\vec{\phi} \leqslant 0$ 的方向 $\vec{d}$ 都不可能是分离方向。这样，我们就可以专注于基点为 $\hat{w} = \vec{\phi}/\|\vec{\phi}\|$ 的半球。而且，将点沿半径扩展到极点处的切面上，还可以避免在半球上的运算。即，我们仅需针对方向 $\vec{d} = x\hat{u} + y\hat{v} + \hat{w}$ 分析 g_0，其中 $\{\hat{u}, \hat{v}, \hat{w}\}$ 构成一个向量右手正交系。定义旋转矩阵 $\mathbf{R} = [\hat{u}|\hat{v}|\hat{w}]$，其各列为指定的向量，$g_0$ 在平面上的限制条件为 $F(x, y) = g_0(\vec{d}) = g_0(\mathbf{R}\vec{\xi})$，其中 $\vec{\xi} = (x, y, 1)$，因此

$$F(x, y) = r_0\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\| + r_1\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\| - \|\vec{\phi}\| \tag{11.43}$$

要确定 $F(x, y) < 0$ 对某些 (x, y) 成立，只需证明 F 的最小值是否为负数。其最小值必定出现在临界点。在这些点上 $\vec{\nabla}F$ 为零或者无定义。任何不满足这个关于 g_0 的不等式限制条件的临界点都被排除，因为 F 可被看做 g_0 对由不等式条件定义的平面上的凸子集的限制条件。我们仅需在该凸子集上计算 F 的最小值，因此位于凸集之外的临界点不满足要求。分析伴随函数 g_1 的对应 $F(x, y)$ 需要使用 $\vec{d} = x\hat{u} + y\hat{v} - \hat{w}$。

8. 对 $F(x, y)$ 的分析

利用 $\partial\mathbf{R}\vec{\xi}/\partial x = \hat{u}$ 和 $\partial\mathbf{R}\vec{\xi}/\partial y = \hat{v}$，$F$ 的偏导数可表示为

$$\frac{\partial F}{\partial x} = \hat{u}\cdot\left(r_0\mathbf{A}_0\frac{\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}}{\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\|} + r_1\mathbf{A}_1\frac{\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}}{\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\|}\right) \quad 和$$

$$\frac{\partial F}{\partial y} = \hat{v}\cdot\left(r_0\mathbf{A}_0\frac{\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}}{\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\|} + r_1\mathbf{A}_1\frac{\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}}{\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\|}\right)$$

如果我们定义 $\mathbf{A} = [\hat{u}|\hat{v}]$，那么方程 $\vec{\nabla}F(x, y) = (0, 0)$ 可概述为

$$\mathbf{A}^{\mathrm{T}}\left(r_0\mathbf{A}_0\frac{\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}}{\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\|} + r_1\mathbf{A}_1\frac{\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}}{\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\|}\right) = \vec{0}$$

定义单位长度向量 $\hat{\eta}_i = \mathbf{A}_i^{\mathrm{T}}\mathbf{R}\vec{\xi}/\|\mathbf{A}_i^{\mathrm{T}}\mathbf{R}\vec{\xi}\|$（$i = 0, 1$）。定义 2×2 矩阵 $\mathbf{B}_i = \mathbf{A}^{\mathrm{T}}\mathbf{A}_i$。要求解的方程为

$$r_0\mathbf{B}_0\hat{\eta}_0 + r_1\mathbf{B}_1\hat{\eta}_1 = \vec{0}, \quad \|\hat{\eta}_0\|^2 = 1 \text{ 和 } \|\hat{\eta}_1\|^2 = 1 \tag{11.44}$$

对于这些方程的任何解 $\hat{\eta}_0$ 和 $\hat{\eta}_1$，$\hat{\eta}_i$ 和 $\mathbf{A}_i^{\mathrm{T}}\mathbf{R}\vec{\xi}$ 也必定指向与它们相同的方向。即

$$\hat{\eta}_0^{\perp} \cdot \mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi} = 0, \quad \hat{\eta}_1^{\perp} \cdot \mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi} = 0, \quad \hat{\eta}_0 \cdot \mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi} > 0 \text{ 和 } \hat{\eta}_1 \cdot \mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi} > 0 \tag{11.45}$$

其中$(a,b)^{\perp}=(b,-a)$。满足这些条件的每一对(x,y)都是满足$\vec{\nabla}F(x,y)=(0,0)$的$F(x,y)$的临界点。然后可以检测临界点，以确定$F(x,y)<0$是否成立，如果成立，则两个圆柱面是分离的。

上述分析$g0(\vec{d}(x,y))=F(x,y)$的算法可概述如下：

（1）利用表示法$\mathbf{R}_i=[\hat{u}_i|\vec{v}_i|\hat{w}_i]$（$i=0,1$），需要计算该算法中用到的不同的点积。它们的 18 个值可抽象表示为$G_0=\mathbf{R}^{\mathrm{T}}\mathbf{R}_0=[g_{ij}^{(0)}]$和$G_1=\mathbf{R}^{\mathrm{T}}\mathbf{R}_1=[g_{ij}^{(1)}]$。

（2）求解$r_0\mathbf{B}_0\hat{\eta}_0+r_1\mathbf{B}_1\hat{\eta}_1=\vec{0}, \|\hat{\eta}_0\|^2=1$和$\|\hat{\eta}_1\|^2=1$中的$\hat{\eta}_0$和$\hat{\eta}_1$。注意，存在多个解对，最明显的是，当$(\hat{\eta}_0,\hat{\eta}_1)$是一个解时，$(-\hat{\eta}_0,-\hat{\eta}_1)$也是一个解。在第 4 步中抽取$(x,y)$时，这样的互反的解对将得到相同的方程系统，因此可以忽略它。

（3）对每一个解对$(\hat{\eta}_0,\hat{\eta}_1)$，求解$\hat{\eta}_0^{\perp}\cdot\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}=0$和$\hat{\eta}_1^{\perp}\cdot\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}=0$中的$\vec{\xi}$。这一组方程也可能具有多个解。

（4）对每一个解对$\vec{\xi}$，校验$\hat{w}_0\cdot\mathbf{R}\vec{\xi}\geqslant 0, \hat{w}_1\cdot\mathbf{R}\vec{\xi}, \hat{\eta}_0\cdot\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}>0$和$\hat{\eta}_1\cdot\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}>0$。

（5）在最后一步中，对从一个有效的$\vec{\xi}$中得到的每一对数(x,y)，都检测$F(x,y)<0$是否成立。如果成立，那么$\vec{d}=\mathbf{R}\vec{\xi}$是分离圆柱面的一个分离方向，同时终止算法。

分析$g_1(x\hat{u}+y\vec{v}-\hat{w})$的算法的前三步与此相同。在第四步和第五步中惟一的不同是求解$\vec{\xi}=(x,y,1)$中的g_0，以及$\vec{\xi}=(x,y,-1)$中的g_1。

9. 求解$\hat{\eta}_i$

由于

$$\mathbf{B}_i=\begin{bmatrix}\hat{u}\cdot\hat{u}_i & \hat{u}\cdot\vec{v}_i\\ \hat{v}\cdot\hat{u}_i & \hat{v}\cdot\vec{v}_i\end{bmatrix}$$

因此

$$\det(\mathbf{B}_i)=(\hat{u}\cdot\hat{u}_i)(\hat{v}\cdot\vec{v}_i)-(\hat{u}\cdot\vec{v}_i)(\hat{v}\cdot\hat{u}_i)=(\hat{u}\times\hat{v})\cdot(\hat{u}_i\times\vec{v}_i)=\hat{w}\cdot\hat{w}_i$$

如果$\det(\mathbf{B}_0)=0$和$\det(\mathbf{B}_1)=0$，那么$\hat{w}$必定同时垂直于$\hat{w}_0$和$\hat{w}_1$。由于$\hat{w}=\vec{\phi}/\|\vec{\phi}\|$，$\vec{\phi}$同时垂直于$\hat{w}_0$和$\hat{w}_1$。注意，$\vec{\phi}=(C_1-(h_1/2)\hat{w}_1)-(C_0+(h_0/2)\hat{w}_0)$，分别取自每一个圆柱面的两个端点之差。因此，连接这两个端点的线段垂直于每一个圆柱面。自己画一幅图形，就可以看出，通过检测方向$\vec{d}=\hat{w}$就能惟一地确定相交 / 分离。注意，由于$\hat{w}\cdot\hat{w}_0=0, \hat{w}\cdot\hat{w}_1=0$和$\hat{w}\cdot\vec{\Delta}=\hat{w}\cdot\vec{\phi}=\|\vec{\phi}\|>0$，因此该方向确实满足上述的不等式。当且仅当$\|\vec{\phi}\|^2>(r_0+r_1)^2$时，这两个圆柱面是分离的。

如果$\det(\mathbf{B}_0)\neq 0$且$\det(\mathbf{B}_1)=0$，那么$\mathbf{B}_1$的列是线性无关的。而且，其中的一个必定是非零向量。否则，$0=(\hat{u}\cdot\hat{u}_1)^2+(\hat{v}\cdot\hat{u}_1)^2=1-(\hat{w}\cdot\hat{u}_1)^2$，即$|\hat{w}\cdot\hat{u}_1|=1$且$\hat{u}_1$为$\hat{w}$或者$-\hat{w}$。类似地，$\vec{v}_1$也为$\hat{w}$或者$-\hat{w}$。这是不可能的，因为$\hat{u}_1$和$\vec{v}_1$是正交的。设$\vec{\psi}$为$\mathbf{B}_1$的非零列。向量$\vec{\zeta}=\vec{\psi}^{\perp}$满足条件$\mathbf{B}_1^{\mathrm{T}}\vec{\zeta}=\vec{0}$，因此

$$0=\vec{\zeta}^{\mathrm{T}}(r_0\mathbf{B}_0\hat{\eta}_0+r_1\mathbf{B}_1\hat{\eta}_1)=r_0(\mathbf{B}_0^{\mathrm{T}}\vec{\zeta})\cdot\hat{\eta}_0$$

如果$\mathbf{B}_0^{\mathrm{T}}\vec{\zeta}=(a,b)$，那么$\hat{\eta}_0=\pm(b,-a)/\sqrt{a^2+b^2}$。向量$\hat{\eta}_1$由$\|\hat{\eta}_1\|=1$和如下的线性方程确定

$$r_1(\mathbf{B}_1^{\mathrm{T}}\vec{\psi})\cdot\hat{\eta}_1=-r_0(\mathbf{B}_0^{\mathrm{T}}\vec{\psi})\cdot\hat{\eta}_0$$

因此，如果圆和直线相交，则$\hat{\eta}_1$就是交点。通过定义$\vec{p}_0=\|\mathbf{B}_0^{\mathrm{T}}\vec{\zeta}\|\hat{\eta}_0$和$\vec{p}_1=\|\mathbf{B}_0^{\mathrm{T}}\vec{\zeta}\|\hat{\eta}_1$，可以避免$\hat{\eta}_0$的规整化。在这种情形中，$\vec{p}_0=(\mathbf{B}_0^{\mathrm{T}}\vec{\zeta})^{\perp}$且$r_1(\mathbf{B}_1^{\mathrm{T}}\vec{\psi})\cdot\vec{p}_1=-r_0(\mathbf{B}_0^{\mathrm{T}}\vec{\psi})\cdot\vec{p}_0$。稍后讨论的$(x,y)$的抽取操作实际上并不要求规整化。计算直线与圆的相交要求求解一个二次方程，因此必须计算一次平方根（或者可以迭代求解该二次方程，以避免求平方根的运算）。类似的方法可应用于$\det(\mathbf{B}_0)=0$和$\det(\mathbf{B}_1)\neq 0$。

如果$\det(\mathbf{B}_0)\neq 0$且$\det(\mathbf{B}_1)\neq 0$，那么$\mathbf{B}_0$是可逆的，并且

$$\hat{\eta}_0=-(r_1/r_0)\mathbf{B}_0^{-1}\mathbf{B}_1\hat{\eta}_1$$

其中$\|\hat{\eta}_0\|=1$且$\|\hat{\eta}_1\|=1$。稍后讨论的(x,y)的抽取操作并不要求$\hat{\eta}_0$和$\hat{\eta}_1$是单位长度的，因此可以改写这 3 个方程，以避免一些除法和规整化运算。将上述方程改写为

$$r_0\det(\mathbf{B}_0)\hat{\eta}_0=-r_1\,\mathrm{Adj}(\mathbf{B}_0)\mathbf{B}_1\hat{\eta}_1$$

定义$\vec{p}_0=r_0\det(\mathbf{B}_0)\hat{\eta}_0$，$\vec{p}_1=r_1\hat{\eta}_1$和$\mathbf{C}=\mathrm{Adj}(\mathbf{B}_0)\mathbf{B}_1$。方程现在可以写成$\vec{p}_0=-\mathbf{C}\vec{p}_1$，$\|\vec{p}_0\|^2=r_0^2\det(\mathbf{B}_0)^2$和$\|\vec{p}_1\|^2=r_1^2$。

关于$\vec{p}_1$的二次方程为$r_0^2\det(\mathbf{B}_0)^2=\vec{p}_1^{\mathrm{T}}\mathbf{C}^{\mathrm{T}}\mathbf{C}\vec{p}_1$和$\|\vec{p}_1\|^2=r_1^2$。因子$\mathbf{C}^{\mathrm{T}}\mathbf{C}=\mathbf{Q}E\mathbf{Q}^{\mathrm{T}}$，其中$E=\mathrm{Diag}(e_0,e_1)$为特征值，而$\mathbf{Q}$的各列为特征向量。设$\vec{\psi}=\mathbf{Q}^{\mathrm{T}}\vec{p}_1$。方程成为$\|\vec{\psi}\|^2=r_1^2$和$r_0^2\det(\mathbf{B}_0)^2=\vec{\psi}^{\mathrm{T}}E\vec{\psi}$。如果$\vec{\psi}=(a,b)$，那么$a^2+b^2=r_1^2$和$e_0a^2+e_1b^2=r_0^2\det(\mathbf{B}_0)^2$。它们是关于两个未知数$a^2$和$b^2$的两个线性方程。其正式解为$a^2=(e_1r_1^2-r_0^2\det(\mathbf{B}_0)^2)/(e_1-e_0)$和$b^2=(r_1^2-e_0r_0^2\det(\mathbf{B}_0)^2)/(e_1-e_0)$。假设两个方程的右式都是非负的，那么可以得到 4 个解(a,b)，$(-a,b)$，$(a,-b)$和$(-a,-b)$，这与我们预期的一致（椭圆与圆的相交）。仅需考虑(a,b)和$(-a,b)$；其余的两个解在抽取(x,y)时并不能产生新的信息。给定$\vec{\psi}$的一个解，对应的用于抽取操作的非规整化向量为$\vec{p}_1=\mathbf{Q}\vec{\psi}$和$\vec{p}_0=-\mathbf{C}\vec{p}_1$。

10. 求解(x,y)

方程（11.45）中的前面两个方程可以写成两个具有两个未知数x和y的两个方程系统，即

$$\mathbf{C}\begin{bmatrix}x\\y\end{bmatrix}=\vec{d}$$

其中$\hat{\eta}_0=(a_0,b_0)$，$\hat{\eta}_1=(a_1,b_1)$，并且

$$\mathbf{C}=\begin{bmatrix}b_0g_{00}^{(0)}-a_0g_{01}^{(0)} & b_0g_{10}^{(0)}-a_0g_{11}^{(0)}\\ b_1g_{00}^{(1)}-a_1g_{01}^{(1)} & b_1g_{10}^{(1)}-a_1g_{11}^{(1)}\end{bmatrix},\quad \vec{d}=\begin{bmatrix}a_0g_{21}^{(0)}-b_0g_{20}^{(0)}\\ a_1g_{21}^{(1)}-b_1g_{20}^{(1)}\end{bmatrix}$$

如果C是可逆的，那么可以求得(x,y)的惟一解。

如果C是不可逆的，那么问题就稍微复杂一些。如果$\mathrm{Adj}(\mathbf{C})\vec{d}\neq\vec{0}$，则系统无解。否则，系统仅有一个独立的方程。由于$\hat{\eta}_0\neq\vec{0}$，而且$\mathbf{A}_0^{\mathrm{T}}\mathbf{R}$是满秩的（秩为 2），因此 3×1 的向量$\mathbf{R}^{\mathrm{T}}\mathbf{A}_0\hat{\eta}_0^{\perp}$不可能为零向量。实际上，$\hat{\eta}_0^{\perp}$是单位长度的，这说明$\mathbf{A}_0\hat{\eta}^{\perp}$是单位长度的。最后，由于$R$是一个旋转矩阵，因此$\mathbf{R}^{\mathrm{T}}\mathbf{A}_0\hat{\eta}_0^{\perp}$是一个单位长度向量。相同的讨论说明$\mathbf{R}^{\mathrm{T}}\mathbf{A}_1\hat{\eta}_1^{\perp}$是一个单位长度向量。这两个条件和该系统具有无数个解的事实，说明$c_{00}^2+c_{01}^2\neq 0$和$c_{10}^2+c_{11}^2\neq 0$。

如果$c_{01}\neq 0$，那么$y=(d_0-c_{00}x)/c_{01}$。将其代入$\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}$，可得

$$\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi} = \frac{(g_{00}^{(0)}g_{11}^{(0)} - g_{01}^{(0)}g_{10}^{(0)})x + (g_{11}^{(0)}g_{20}^{(0)} - g_{10}^{(0)}g_{21}^{(0)})}{a_0 g_{11}^{(0)} - b_0 g_{10}^{(0)}} =: (\alpha_0 x + \beta_0)\hat{\eta}_0$$

α_0的分子为$\det(\mathbf{B}_0)$。如果$c_{01}=0$，那么$c_{00}\neq 0$，并且可以得到关于y的相似的表达式来表示$\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}$，即$\alpha_0' y+\beta_0'$，其中$\alpha_0'$的分子也是$\det(\mathbf{B}_0)$。类似地，如果$c_{11}\neq 0$，那么$y=(d_1-c_{10}x)/c_{11}$，并且

$$\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi} = \frac{(g_{00}^{(1)}g_{11}^{(1)} - g_{01}^{(1)}g_{10}^{(1)})x + (g_{11}^{(1)}g_{20}^{(1)} - g_{10}^{(1)}g_{21}^{(1)})}{a_1 g_{11}^{(1)} - b_1 g_{10}^{(1)}} =: (\alpha_1 x + \beta_1)\hat{\eta}_1$$

α_1的分子为$\det(\mathbf{B}_1)$。如果$c_{11}=0$，那么$c_{10}\neq 0$，并且可以得到关于y的相似的表达式来表示$\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}$，即$\alpha_1' y+\beta_1'$，其中$\alpha_1'$的分子也是$\det(\mathbf{B}_1)$。

如果$c_{01}\neq 0$和$c_{11}\neq 0$，那么$F(x,y)$可简化为

$$F(x,y) = r_0|\alpha_0 x+\beta_0| + r_1|\alpha_1 x+\beta_1| - |\vec{\phi}|$$

如果$\alpha_0\neq 0$和$\alpha_1\neq 0$，那么可在$x=-\beta_0/\alpha_0$或者$x=-\beta_1/\alpha_1$处取得F的最小值。注意，第一个x将使$\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}=\vec{0}$，此时对应的方向必定为$\vec{d}=\hat{w}_0$。第二个$x$将使$\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}=\vec{0}$，此时对应的方向必定为$\vec{d}=\hat{w}_1$。前面已经检测过这两个方向，因此可以忽略这种情形。如果$\alpha_0\neq 0$和$\alpha_1=0$，那么可在$x=-\beta_0/\alpha_0$处取得F的最小值。对应的方向必定为$d=\hat{w}_0$，前面也已经处理过。可对$\alpha_0=0$和$\alpha_1\neq 0$进行相同的讨论。最后的一种情形是$\alpha_0=\alpha_1=0$，此时$\det(\mathbf{B}_0)=\det(\mathbf{B}_1)=0$，这种情形前面也已经处理过。因此，在实现时可以忽略这些情形。当$c_{00}\neq 0$和$c_{10}\neq 0$时，可以进行相似的讨论，而且，$F(x,y)$可简化为

$$F(x,y) = r_0|\alpha_0' y+\beta_0'| + r_1|\alpha_1' y+\beta_1'| - \|\vec{\phi}\|$$

由于已在另一次分离检测中处理过这些情形，因此在实现时都可以忽略它们。最后，如果存在一个x和y的混合项，

$$F(x,y) = r_0|\alpha_0 x+\beta_0| + r_1|\alpha_1' y+\beta_1'| - \|\vec{\phi}\|$$

或者

$$F(x,y) = r_0|\alpha_0' y+\beta_0'| + r_1|\alpha_1 x+\beta_1| - \|\vec{\phi}\|$$

那么对每一维分别进行最小值处理，但是与前面一样，其他的检测涵盖了这些情形。结论就是，当C不可逆时，实现时并不需要做任何事情。

11．检测$F(x,y)<0$的快速方法

可以忽略方程（11.43）中的两个平方根计算，即$\|\mathbf{A}_i^{\mathrm{T}}\mathbf{R}\vec{\xi}\|$。检测$F(x,y)<0$等价于

$$r_0\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\| + r_1\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\| < \|\vec{\phi}\|$$

对该不等式两边平方，并移项，可得到如下的测试条件

$$2r_0 r_1\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\|\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\| < \|\vec{\phi}\|^2 - r_0^2\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\|^2 - r_1^2\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\|^2 =: \rho$$

如果$\rho\leqslant 0$，那么必须满足$F(x,y)\geqslant 0$，并且不需做其他的工作。如果$\rho>0$，那么两边再平方一次，可得到如下的测试条件

$$4r_0^2 r_1^2\|\mathbf{A}_0^{\mathrm{T}}\mathbf{R}\vec{\xi}\|^2\|\mathbf{A}_1^{\mathrm{T}}\mathbf{R}\vec{\xi}\|^2 < \rho^2$$

11.12.10　线形对象与环面的相交

在本节中我们讨论线形对象与环面的相交问题，如图 11.74 所示。线形对象（一般）定义为一个基点和一个方向：

$$\mathcal{L}(t) = P + t\vec{d} \tag{11.46}$$

线段则用两个点 P_0 和 P_1 来定义，我们可以设 $\vec{d} = P_1 - P_0$。

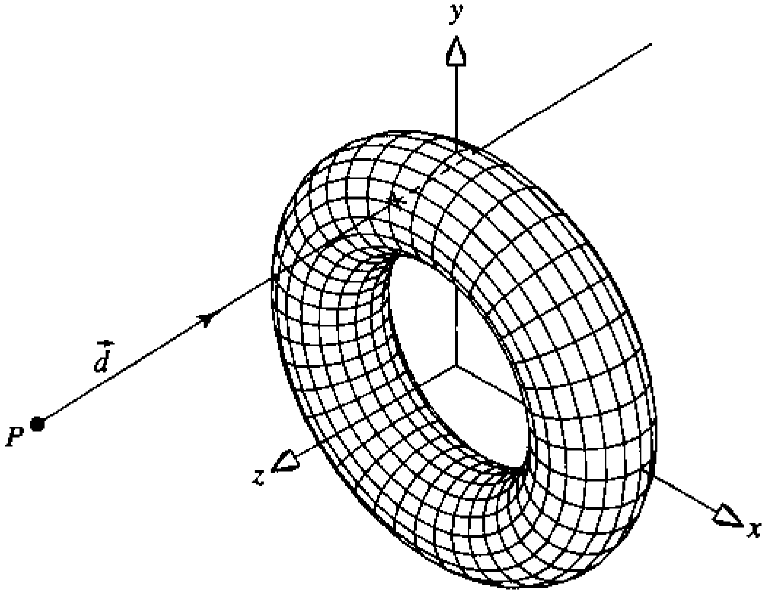


图 11.74　线形对象与环面的相交

一个环面可以隐含地定义为

$$(x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2) = 0 \tag{11.47}$$

上式定义了一个中心位于原点，并位于 XY 平面上的一个环面，其主半径为 R，次半径为 r。

如果我们将方程（11.46）代入方程（11.47），可以得到一个关于 t 的二次方程，其形式为

$$c_4t^4 + c_3t^3 + c_2t^2 + c_1t + c_0 = 0 \tag{11.48}$$

其中

$$c_4 = (\vec{d} \cdot \vec{d})^2$$

$$c_3 = 4(P \cdot \vec{d})(\vec{d} \cdot \vec{d})$$

$$c_2 = 2(\vec{d} \cdot \vec{d})((P \cdot P) - (R^2 + r^2)) + 4(P \cdot \vec{d})^2 + 4R^2\vec{d}_z^2$$

$$c_1 = 4(P \cdot \vec{d})((P \cdot P) - (R^2 + r^2)) + 8R^2P_z\vec{d}_z$$

$$c_0 = ((P \cdot P) - (R^2 + r^2)) - 4R^2(r^2 - P_z^2)$$

可以利用一种求根方法来求解该方程，参见 A.5 节。

如果是为了射线跟踪的目的而计算相交，那么想要的交点（最多可能存在4个交点）将是最接近射线原点的那个交点。此外还需要位于该点的曲面法线，以及"纹理坐标"和偏导数。通过计算方程（11.48）关于 x，y和z的偏导数，就能计算出这条法线。可是，还有一种更直接的方法。考虑一个环面的"切面"，如图11.75所示。由于其切面是一个圆，圆上任意点X的法线$\vec{n}$就是向量$(X-C)$。然而，可以很容易地求得C：将交点X投影到XY平面上：$X'=[X_x\quad X_y\quad 0]$，那么C就是沿着向量$(X'-O)$与原点$O=[0\quad 0\quad 0]$的距离为R的点。注意，除非$r=1$，否则向量$\vec{n}=X-C$并不是规整化的。

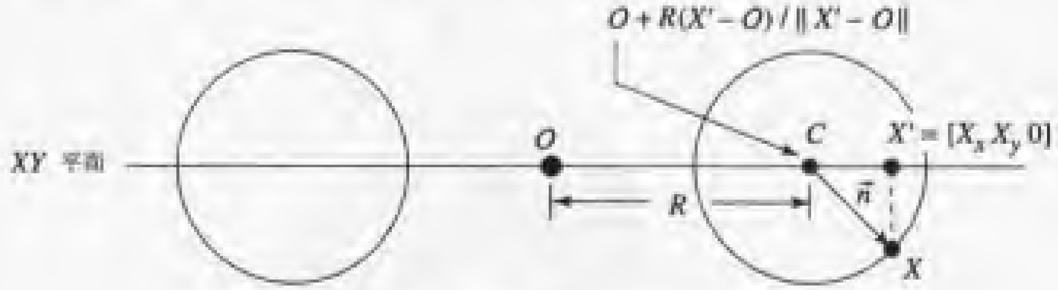


图 11.75　计算环面上某一（相交）点的法线

为了计算纹理坐标，我们定义u方向为（沿z轴往下看）从x轴开始的逆时针方向。v方向为从里面开始绕环面的"扫管"的圆周方向。我们先讨论参数u。考虑切面视图（这次为沿XZ平面的切面），如图11.76所示。

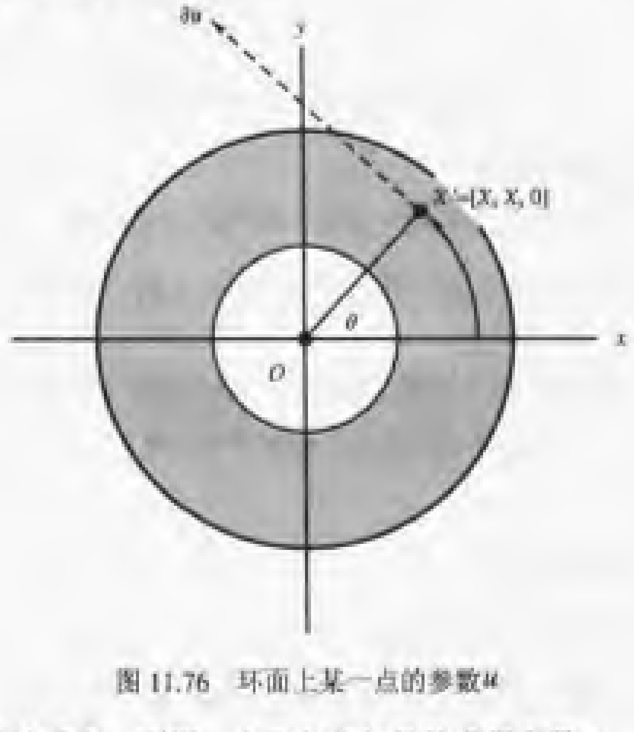


图 11.76　环面上某一点的参数u

从图中可以很容易地看出，利用一点三角几何就能求得参数u：

$$r_u=\|X'-O\|$$

$$\cos(\theta)=\frac{X_x}{r_u}$$

$$\sin(\theta) = \frac{X_y}{r_u}$$

$$u = \begin{cases} \frac{\arccos(\theta)}{2\pi} & \text{如果 } \sin(\theta) \geq 0 \\ 1 - \frac{\arccos(\theta)}{2\pi} & \text{如果 } \sin(\theta) < 0 \end{cases}$$

对于v方向，考虑图 11.77。为了求得$\cos(\phi)$，我们需要知道直角三角形中这个角的相邻直角边和斜边——它们分别为$\|X' - \mathcal{O}\| - R$和r。注意，如果我们希望参数原点位于画面的里面，那么我们就需要反转（即取负）其余弦。为了求得$\sin(\phi)$，我们需要知道该角的对边，也就是X_z。因此，我们有

$$r_v = \|X' - \mathcal{O}\|$$

$$\cos(\phi) = \frac{-(r_v - R)}{r}$$

$$\sin(\phi) = \frac{X_z}{r_v}$$

$$u = \begin{cases} \frac{\arccos(\phi)}{2\pi} & \text{如果 } \sin(\theta) \geq 0 \\ 1 - \frac{\arccos(\phi)}{2\pi} & \text{如果 } \sin(\theta) < 0 \end{cases}$$

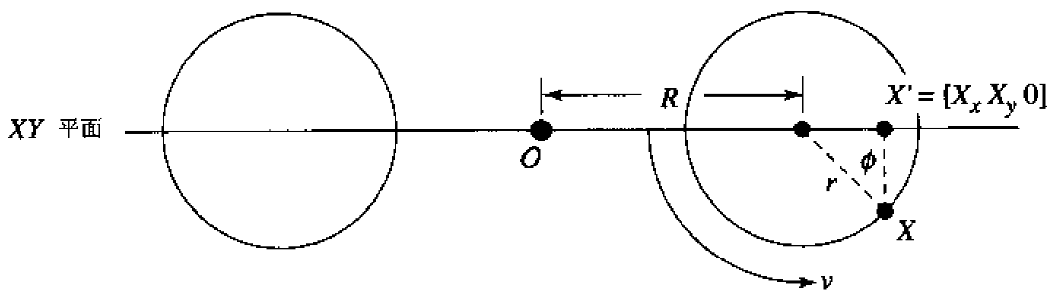


图 11.77 环面上某一点的参数 v

它的偏导数的计算非常简单：观察图 11.76，可以明显地看出

$$\partial u = (X' - \mathcal{O})^{\perp}$$

利用一种与计算$\vec{n}$时相似的方法，我们就能确定∂v，但是，实际上，我们可以利用一种更高效的方法，只要注意到

$$\partial v = \vec{n} \times \partial u$$

注意，偏导数是在环面的局部空间上计算的。需要将它们变换回世界空间中，并在世界空间中规整化。

第 12 章　其他三维问题

本章包括一系列涉及三维直线、平面、四面体和三维圆的问题。它们中大部分都是经常（至少是有时）遇到的问题，另外一些虽然不是很常见，但可以用来介绍如何用不同的技术来解决新的问题。

12.1　点在平面上的投影

在本节中，我们讨论一个点 Q 在一个平面 $\mathcal{P}$ 上的投影，其中 $\mathcal{P}$ 定义为 $ax + by + cz + d = 0$（或者 $P \cdot \vec{n} + d = 0$），如图 12.1 所示。根据定义，Q 与其投影 Q' 之间的线段平行于平面的法线 $\vec{n}$。设这两个点之间的（有符号）距离为 r，可得

$$Q = Q' + \frac{r\vec{n}}{\|\vec{n}\|} \tag{12.1}$$

如果将方程的两边与 $\vec{n}$ 点积，可得

$$\begin{aligned} Q \cdot \vec{n} &= \left(Q' + \frac{r\vec{n}}{\|\vec{n}\|}\right) \cdot \vec{n} \\ &= Q' \cdot \vec{n} + \frac{r\vec{n}}{\|\vec{n}\|} \cdot \vec{n} \end{aligned} \tag{12.2}$$

然而 $Q' \cdot \vec{n} = -d$，由于根据定义，Q' 位于平面 $\mathcal{P}$ 上，并且 $\vec{n} \cdot \vec{n} = \|\vec{n}\|^2$，因此 $\frac{r\vec{n}}{\|\vec{n}\|} \cdot \vec{n} = r\|\vec{n}\|$。如果将其代入方程（12.2），并求解 r，则可得

$$r = \frac{Q \cdot \vec{n} + d}{\|\vec{n}\|} \tag{12.3}$$

然后我们可以对方程（12.1）进行移项，并将其代入方程（12.3）：

$$\begin{aligned} Q' &= Q - \frac{r\vec{n}}{\|\vec{n}\|} \\ &= Q - \frac{\frac{Q \cdot \vec{n} + d}{\|\vec{n}\|}\vec{n}}{\|\vec{n}\|} \\ &= Q - \frac{Q \cdot \vec{n} + d}{\|\vec{n}\|\|\vec{n}\|}\vec{n} \\ &= Q - \frac{Q \cdot \vec{n} + d}{\vec{n} \cdot \vec{n}}\vec{n} \end{aligned}$$

如果该平面方程是规整化的，我们就有 $\|\hat{n}\| = 1$，因而可以避免其中的除法运算：

$$Q' = Q - (Q \cdot \hat{n} + d)\,\hat{n}$$

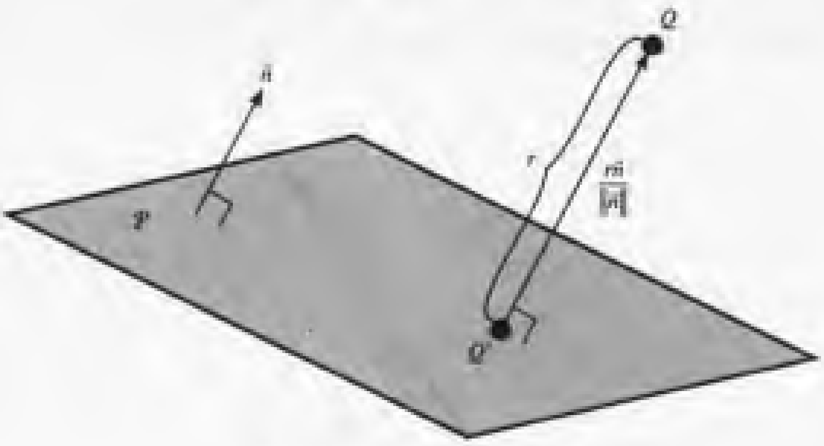


图 12.1 点在平面上的投影

12.2 向量在平面上的投影

向量 $\vec{v}$ 在平面 $\mathcal{P}: P \cdot \hat{n} + d = 0$ 上的投影可以表示如下（如图 12.2 所示）：

$$\vec{w} = \vec{v} - (\vec{v} \cdot \hat{n})\hat{n} \tag{12.4}$$

或者

$$\vec{w} = \vec{v} - \frac{\vec{v} \cdot \vec{n}}{\|\vec{n}\|^2}\vec{n}$$

其中假设平面法线不是单位长度的。

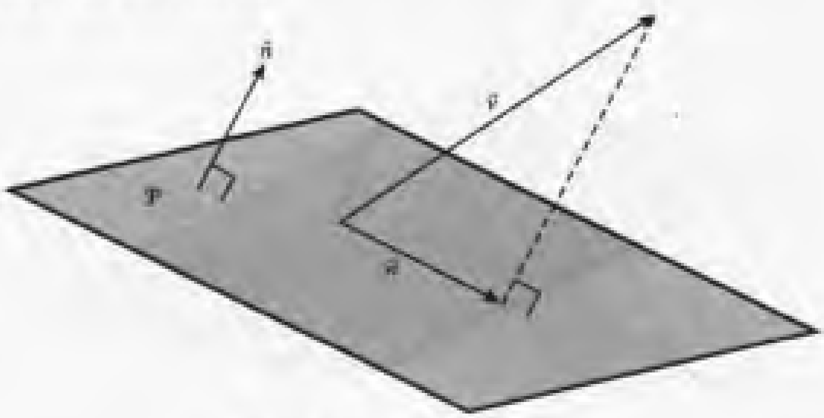


图 12.2 向量在平面上的投影

为了理解为什么会这样，观察一下图 12.3。在图中，我们可以看到 $\vec{v}$ 投影到 $\hat{u}$ 上。向量 $\vec{v}$ 可以分解为 $\vec{v} = \vec{v}_{\perp} + \vec{v}_{\parallel}$，即分别为相对于 $\hat{u}$ 的平行分量和垂直分量。根据点积的定义，$\vec{v}_{\parallel}$ 的长度为 $\vec{v} \cdot \hat{u}$，而且由于两个分量的和为 $\vec{v}$，因此有

$$\vec{v}_{\parallel} = (\vec{v} \cdot \hat{u})\hat{u} \tag{12.5}$$

$$\vec{v}_{\perp} = \vec{v} - (\vec{v} \cdot \hat{u})\hat{u} \tag{12.6}$$

如果我们将图 12.3 中的向量 $\vec{u}$ 看做图 12.2 中的平面法线 $\hat{n}$，那么就可以看出 $\vec{w}=\vec{v}_{\perp}$，因而可以直接从方程（12.5）中得出方程（12.4）。这里的要点是认识到对于任何垂直于 $\hat{n}$ 的平面，$\vec{v}$ 在 $\mathcal{P}$ 上的投影都相同，因此这样就能理解为什么 $\mathcal{P}$ 的平面方程中的 d 项没有出现在解中。

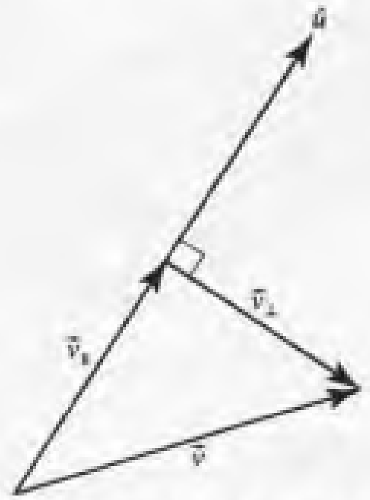


图 12.3　一个向量在另一个向量上的投影

12.3　直线与平面的夹角

可以利用多种方法来计算直线 $\mathcal{L}(t)=P_0+t\vec{d}$ 与平面 $\mathcal{P}: P_1\cdot\vec{n}+d=0$ 之间的夹角，具体方法取决于直线、平面的规整性（如图 12.4 所示）。计算直线与平面法线之间的夹角 ϕ 的公式如表 12.1 所示。直线与平面之间的夹角为 $\theta=\frac{\pi}{2}-\phi$。

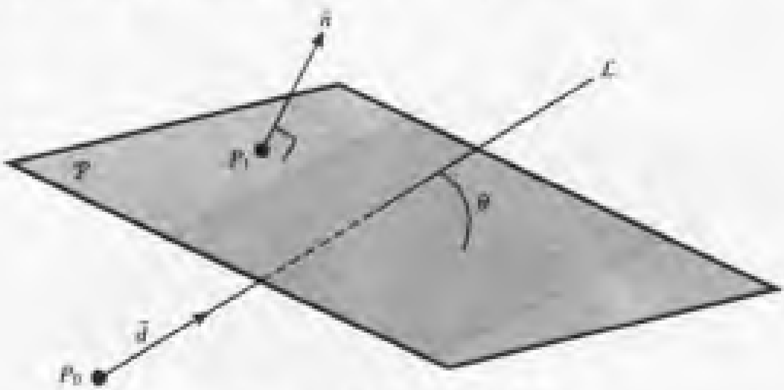


图 12.4　直线与平面的夹角

表 12.1　计算 ϕ 的公式

平　面	直　线	
	规整化的	不规整化的
规整化的	$\phi=\arccos(\hat{n}\cdot\hat{d})$	$\phi=\arccos\frac{\hat{n}\cdot\vec{d}}{\|\vec{d}\|}$
不规整化的	$\phi=\arccos\frac{\vec{n}\cdot\hat{d}}{\|\vec{n}\|}$	$\phi=\arccos\frac{\vec{n}\cdot\vec{d}}{\|\vec{n}\|\|\vec{d}\|}$

12.4 两平面之间的夹角

可以简单地通过它们的法线之间的夹角来计算两个平面 $\mathcal{P}_0:\{P_0, \hat{n}_0\}$ 和 $\mathcal{P}_1:\{P_1, \hat{n}_1\}$ 之间的夹角（如图 12.5 所示）。与直线和平面之间的夹角一样，也可用多种方法来计算两个平面之间的夹角 θ，取决于 $\mathcal{P}_0$，P_1的规整性（如表 12.2 所示）。

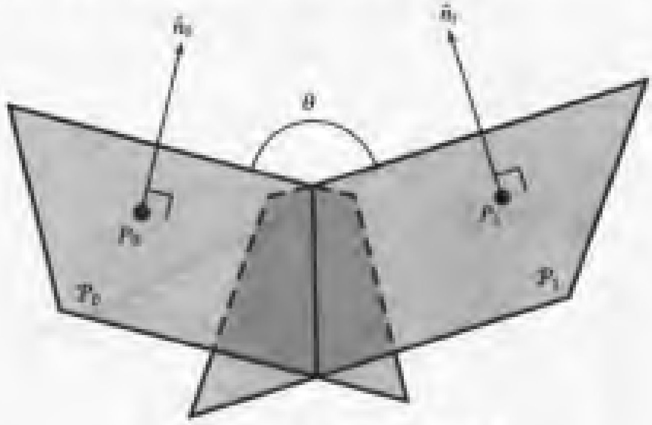


图 12.5　两个平面之间的夹角

表 12.2　计算 θ 的公式

$\mathcal{P}_0$	$\mathcal{P}_1$	
	规整化的	不规整化的
规整化的	$\theta = \arccos(\hat{n}_0 \cdot \hat{n}_1)$	$\theta = \arccos \frac{\hat{n}_0 \cdot \vec{n}_1}{\|\vec{n}_1\|}$
不规整化的	$\theta = \arccos \frac{\vec{n}_0 \cdot \hat{n}_1}{\|\vec{n}_0\|}$	$\theta = \arccos \frac{\vec{n}_0 \cdot \vec{n}_1}{\|\vec{n}_0\|\|\vec{n}_1\|}$

12.5 以一条直线为法线并通过一给定点的平面

假设我们有一条直线$\mathcal{L}(t) = P + t\vec{d}$和一个点$Q$（如图 12.6 所示）。平面的经过$Q$的法线$\mathcal{L}$为

$$d_x x + d_y y + d_z z - (d_x Q_x + d_y Q_y + d_z Q_z) = 0$$

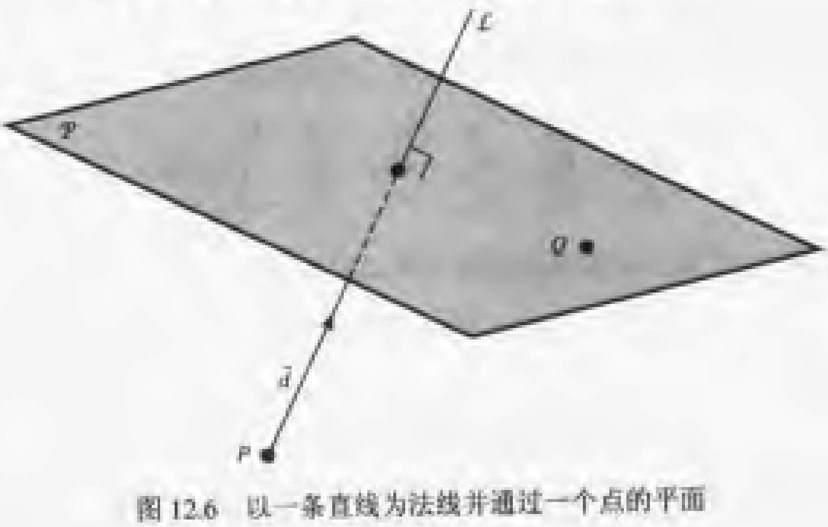


图 12.6　以一条直线为法线并通过一个点的平面

$\mathcal{P}$ 的法线显然就是$\mathcal{L}$的方向向量。$\mathcal{P}$ 的方程的系数d可能并不那么明显。然而，考虑图 12.7 中显示的“切面”图，如果我们画出起始于原点$\mathcal{O}$的$\mathcal{L}$的方向向量，并考虑向量$\overrightarrow{\mathcal{O}Q}$，很明显，$\mathcal{P}$ 的方程的系数d就是$\overrightarrow{\mathcal{O}Q}$在$\vec{d}$上的投影，因而我们有

$$d = -(\overrightarrow{\mathcal{O}Q} \cdot \vec{d})$$

其等价的分量表示法为

$$-(d_x Q_x + d_y Q_y + d_z Q_z)$$

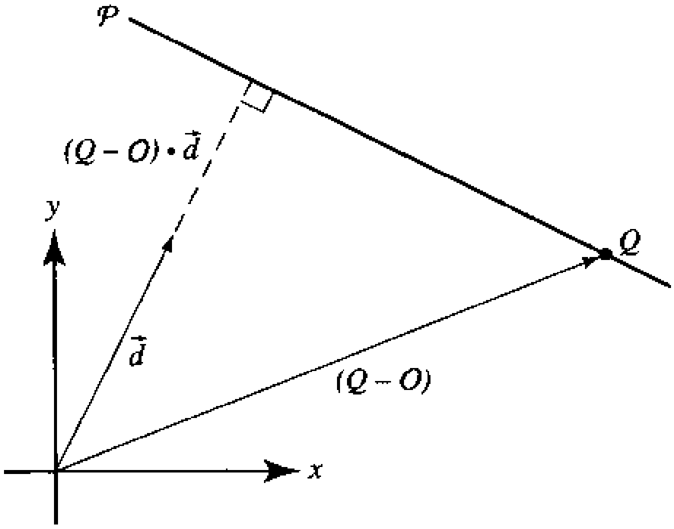


图 12.7　计算平面的距离系数

不论$\mathcal{L}$是规整化的还是不规整化的，该方程都成立，并且当且仅当$\|\vec{d}\|=1$时，$\mathcal{P}$ 的方程才是规整化的。

12.6　三点决定的平面

给定三个点P_0, P_1和P_2，它们不重合，并且三者不共线，我们可以非常直接地导出这三个点所确定的平面（如图 12.8 所示）。实际上，我们可以基于这三个点非常直接地导出平面的多种表示法中的任何一种。

- 隐含形式：经过三个点的平面的隐含形式方程满足

$$\begin{vmatrix} x - P_{0,x} & y - P_{0,y} & z - P_{0,z} \\ P_{1,x} - P_{0,x} & P_{1,y} - P_{0,y} & P_{1,z} - P_{0,z} \\ P_{2,x} - P_{0,x} & P_{2,y} - P_{0,y} & P_{2,z} - P_{0,z} \end{vmatrix} = 0$$

  如果我们将该式乘出来就能得到如下形式的方程

$$ax + by + cz + d = 0$$

- 参数形式：

$$\mathcal{P}(s,t) = P_0 + s(P_1 - P_0) + t(P_2 - P_0)$$

- 显式形式：这种形式要求我们在平面上指定一个点、一条法线，以及第三个参数d

（它表示与原点的垂直距离）。我们有三个点，可以从中选取一个，并能通过计算两对点之间向量的叉积来计算平面的法线（前面已经介绍过）。任意选择平面上的一个点 P_0，并且法线 $\vec{n}$ 为 $(P_1 - P_0) \times (P_2 - P_0)$，可得

$$P_0 \cdot \vec{n} + d = 0$$

可以很简单地计算出 $d = -(P_0 \cdot \vec{n})$。

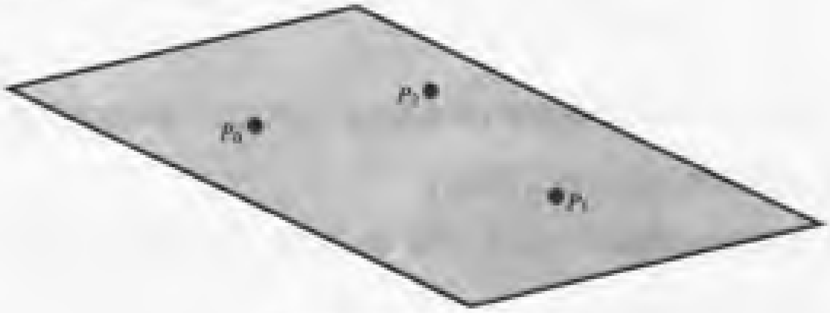


图 12.8　三点定义一个平面

12.7　两条直线之间的夹角

给定两条直线 $\mathcal{L}_0(t) = P_0 + t\vec{d}_0$ 和 $\mathcal{L}_1(t) = P_1 + t\vec{d}_1$，它们之间的夹角可以利用点积与角度之间的关系来计算，因此，我们有

$$\theta = \arccos \frac{\vec{d}_0 \cdot \vec{d}_1}{\|\vec{d}_0\|\|\vec{d}_1\|} \tag{12.7}$$

如果两条直线的方程都是规整化的，上式可以简化为

$$\theta = \arccos(\hat{d}_0 \cdot \hat{d}_1)$$

对该问题来说，直线的“方向”不一定是很重要的，特别是当该直线方程定义的是一条直线，而不是一条射线或者线段时。上式计算得到的两条直线之间的角度沿相同的方向“运动”。例如，如果直线是平行的，并且 $\vec{d}_0$ 和 $\vec{d}_1$ 指向相同的方向，那么 $\theta = 0$；否则 $\theta = \pi$（180°）。在图 12.9 中，用方程（12.7）可计算得到 $\theta \approx 46°$，但是如果我们反转其中一条直线的方向，如图 12.10 所示，那么计算得到的结果为 $\theta \approx 134°$。

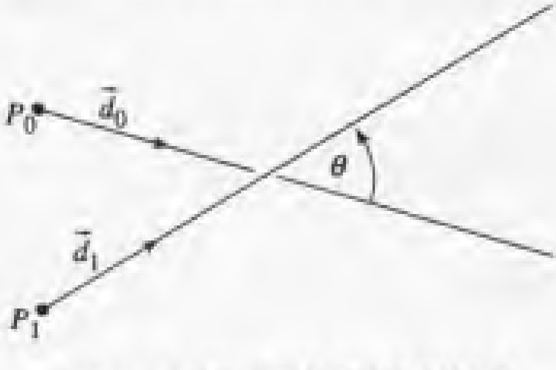


图 12.9　3D 空间中两条直线所成的角度

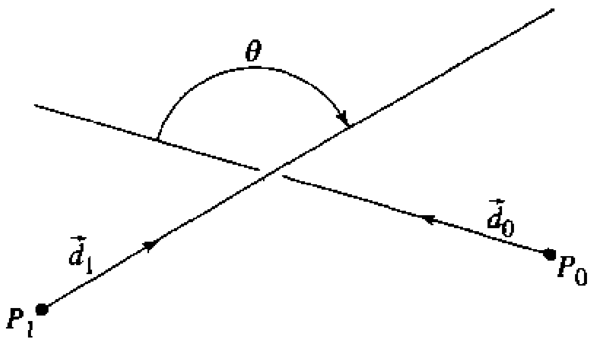


图 12.10　3D 空间中两条直线所成的角度（反转其中一条直线的方向）

如果直线都不是规整化的，那么伪码为

```
float Line3DLine3DAngle(Line l1, Line l2, boolean acuteAngleDesired)
{
    float denominator;

    denominator = Dot(l1.direction, l1.direction) *
                  Dot(l2.direction, l2.direction);
    if (denominator < epsilon) {
       // One or both lines are degenerate,
       // deal with in application-dependent fashion
    } else {
        float angle;

        angle = Acos(Dot(l1.direction, l2.direction) /
                Sqrt(denominator);
        if (acuteAngleDesired && angle > Pi/2) {
            return Pi - angle;
        } else {
            return angle;
        }
    }
}
```

如果直线都是规整化的，那么伪码为

```
float Line3DLine3DAngle(Line l1, Line l2, boolean acuteAngleDesired)
{
    float angle;

    angle = Acos(Dot(l1.direction, l2.direction));
    if (acuteAngleDesired && angle > Pi/2) {
        return Pi - angle;
    } else {
        return angle;
    }
}
```

第 13 章　关于计算几何学的话题

计算几何学领域是一个非常大的领域，也是最近发展最快的前沿领域之一。本章并非关于这一领域的全面介绍，涉及的内容包括二维和三维空间中的空间分区二叉树（BSP）、点在多边形内和点在多面体内的检测、有限点集的凸包、二维和三维空间中的德洛奈三角剖分、将多边形分解为凸片或三角形、二维空间中圆或有向箱的包含点集、三维空间中球或有向箱的包含点集、多边形的面积计算，以及多面体的体积计算等。

我们的重点当然是实现不同主题的各种算法。然而，我们还关注在浮点数系统中的数值计算问题。其间还不断出现一些特定的主题，讨论了确定点何时共线、共面或共球的问题。当采用的计算系统是基于整数算术的系统时，这类计算很简单，但是，当使用浮点算术时，却非常容易出问题。

13.1　二维空间分区二叉树

Fuchs，Kedem 和 Naylor（1979，1980）首先提出利用二叉树来对空间进行分区的方法，这种方法在许多的应用中都非常有效。

考虑平面上一条方程为$\vec{n}\cdot X-c=0$的直线。这条直线将平面划分为两个半平面。$\vec{n}$所指向的直线的一侧叫做这条直线的正侧；另一侧叫做这条直线的负侧。如果X位于正侧，则$\vec{n}\cdot X-c>0$，因此我们用术语“正侧”来称呼它。如果X位于负侧，则$\vec{n}\cdot X-c<0$。位于直线上的点X刚好满足$\vec{n}\cdot X-c=0$。

每一个半平面都可以用平面上的另外一条直线来进行分区。得到的正的和负的区域又可以进一步被分区。用二叉树来表示得到的分区结果，二叉树的每一个节点都表示一条分区直线。一个节点的左子树表示该节点所表示的分区直线的正侧；一个节点的右子树表示该节点所表示的分区直线的负侧。二叉树的叶子节点表示分区得到的凸区域。图 13.1 显示了这种表示方法。其中的正方形用来表示一个平面。分区直线用P来标记，而凸区域用C来标记。

13.1.1　多边形的空间分区二叉树表示

一棵空间分区二叉树表示一个平面的分区，然而也可以用它来将多边形划分成子多边形。在许多方面都可以利用这种分解。这种树支持点在多边形内的检测，我们将在本节的后面讨论这一问题。点在多边形内的其他检测算法将在 13.3 节中介绍。一棵空间分区二叉树表示将一个多边形分解为三角形的一般方法——在 13.9 节中研究了这种思想。最后，多边形的空间分区二叉树表示可用于支持对多边形的布尔运算——在 13.5 节中研究了这种思想。

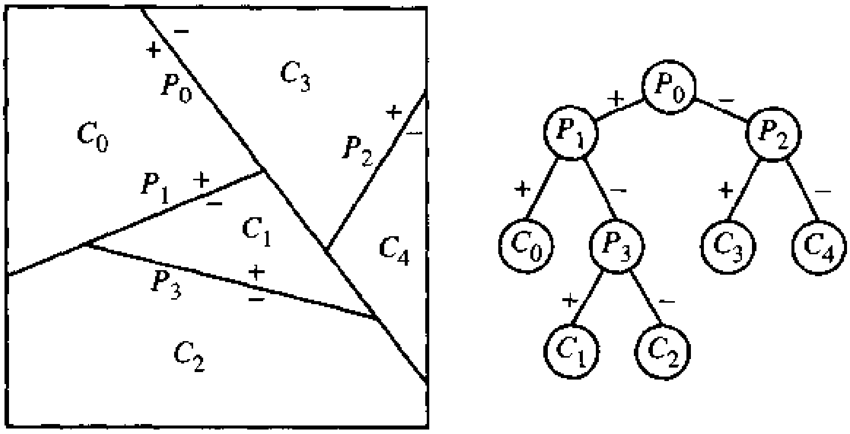


图 13.1　平面的空间分区二叉树分区

对一个多边形构建一棵空间分区二叉树的最简单的方法是建立一系列节点，使每一个节点都表示一条包含多边形的一条边的分区直线。多边形的其他边在节点上被分区直线所分解。位于一条直线的正侧的任何子边都被送往正子树，重复这种处理。位于一条直线的负侧的任何子边都被送往负子树，也重复这种处理。多边形的另一条边也可能完全位于分区直线上。这种边叫做一致边，也存放在表示分区直线的节点中。在这种构建中，至少有一条多边形的边包含于分区直线中。并不要求分区直线包含多边形的边。这本节的稍后部分我们将再次讨论这种思想。下面列出了对一个多边形构建一棵空间分区二叉树的伪码。顶层调用的输入参数列表是来自多边形的边的集合，假设它是非空的。

```
BspTree ConstructTree(EdgeList L)
{
    T = new BspTree;

    // use an edge to determine the splitting line for the tree node
    T.line = GetLineFromEdge(L.first);  // Dot(N, X) - c = 0

    EdgeList posList, negList;  // initially empty lists
    for (each edge E of L) {
        // Determine how edge and line relate to each other.  If the edge
        // crosses the line, the subedges on the positive and negative
        // side of the line are returned.
        type = Classify(T.line, E, SubPos, SubNeg);

        if (type is CROSSES) {
            // Dot(N, X) - c < 0 for one vertex, Dot(N, X) - c > 0
            // for the other vertex
            posList.AddEdge(SubPos);
            negList.AddEdge(SubNeg);
        } else if (type is POSITIVE) {
            // Dot(N, X) - c >= 0 for both vertices, at least one positive
            posList.AddEdge(E);
        } else if (type is NEGATIVE) {
            // Dot(N, X) - c <= 0 for both vertices, at least one negative
            negList.AddEdge(E);
```

```
        } else {
            // type is COINCIDENT
            // Dot(N, X) - c = 0 for both vertices
            T.coincident.AddEdge(E);
        }
    }

    if (posList is not empty)
        T.posChild = ConstructTree(posList);
    else
        T.posChild = null;
    if (negList is not empty)
        T.negChild = ConstructTree(negList);
    else
        T.negChild = null;

    return T;
}
```

函数GetLineFromEdge产生一条直线，其法线向量指向位于指定的边多边形的外部区域。其他的一致边的法线方向与这条直线的法线方向并不一定相同。图 13.2 显示了这种情形。

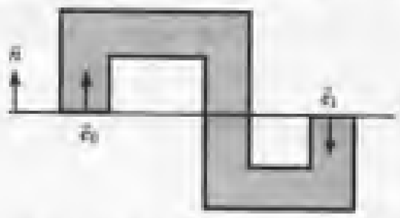


图 13.2 两条一致边具有相反方向的法线的分区线

函数Classify用于寻找当前的边与节点直线的交点。如果存在一个交点为这条边的内点，就返回正的和负的子边。下面的情形是可能的，即这条边的一个端点位于这条直线上，而另一个端点不位于这条直线上。在这种情形中，这条边既可归类为正边，也可归类为负边。当然，这条边也可以完全位于这条直线的任何一侧，而与这条直线根本不相交，在这种情形中，这条边归类为正边或者负边。最后，这条边也可以完全位于分区直线上，在这种情形中，这条边归类为一致边。该函数的伪码如下

```
int Classify(Line L, Edge E, Edge SubPos, Edge SubNeg)
{
    d0 = Dot(L.normal, E.V(0) - L.origin);
    d1 = Dot(L.normal, E.V(1) - L.origin);
    if (d0 * d1 < 0) {
        // edge crosses line
        t = d0 / (d0 - d1);
        I = E.V(0) + t * (E.V(1) - E.V(0));
        if (d1 > 0) {
            SubNeg = Edge(E.V(0), I);
            SubPos = Edge(I, E.V(1));
        } else {
            SubPos = Edge(E.V(0), I);
```

```
                SubNeg = Edge(I, E.V(1));
            }

            return CROSSES;
        } else if (d0 > 0 or d1 > 0) {
            // edge on positive side of line
            return POSITIVE;
        } else if (d0 < 0 or d1 < 0) {
            // edge on negative side of line
            return NEGATIVE;
        } else {
            // edge is contained by the line
            return COINCIDENT;
        }
    }
```

由于浮点数的舍入误差，可能出现 $d_0d_1 < 0$，而 t 接近于 0（或者 1），并可当成 0（或者 1）。Classify 的实现应该包含对这种情形的处理，以避免出现如下的情形：两条边相交于一个顶点，第一条边作为分区直线，而第二条边在数值上表现为与这条直线交叉，这样就导致一次不应该出现的分解。

【实例】 图 13.3 显示了一个倒置的 L 形多边形，它具有 10 个顶点和 10 条边。这些顶点的索引为 0~9。这些边为 (9, 0) 和 $(i, i+1)$（$0 \leqslant i \leqslant 8$）。我们一次构建一条边的空间分区二叉树。在每一步中，分区直线都用点线来表示，正侧区域显示为白色，而负侧区域显示为灰色。

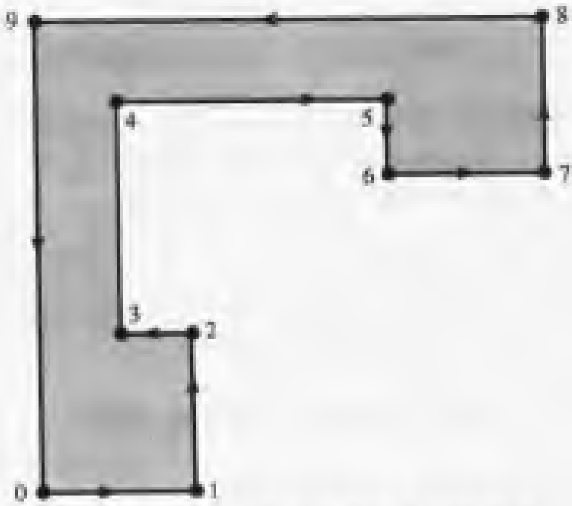


图 13.3　构造一棵 BSP 树的多边形例子

要处理的第一条边是 (9, 0)。图 13.4 显示了用包含这条边的点线对平面进行分区的情形，其中显示了这颗树的根节点(r)。边 (9, 0) 是这个节点的一部分（这条边定义了分区直线），也是分区得到的正边(p)和负边(n)的一部分。在这种情形中，所有剩余的边都位于这条直线的负侧。

要处理的下一条边是(0, 1)。图 13.5 显示了空间分区二叉树在处理这条边之后的状态。

要处理的下一条边是(1, 2)。图 13.6 显示了空间分区二叉树在处理这条边之后的状态。

这条边同时分解边(4, 5)和(8, 9)，产生了两个新的顶点，在图中标示为 10 和 11。

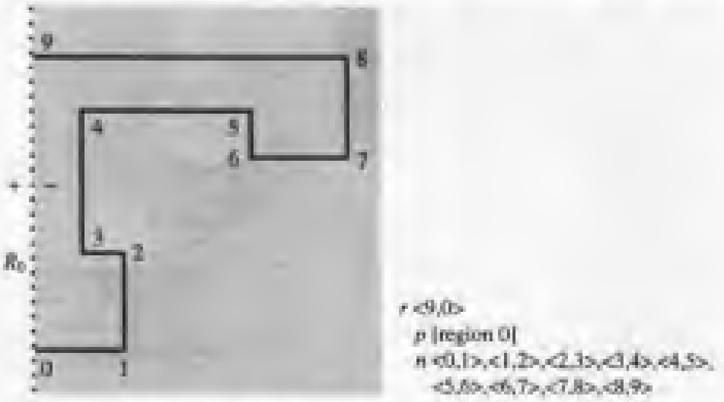


图 13.4　处理边(9, 0)后的当前状态

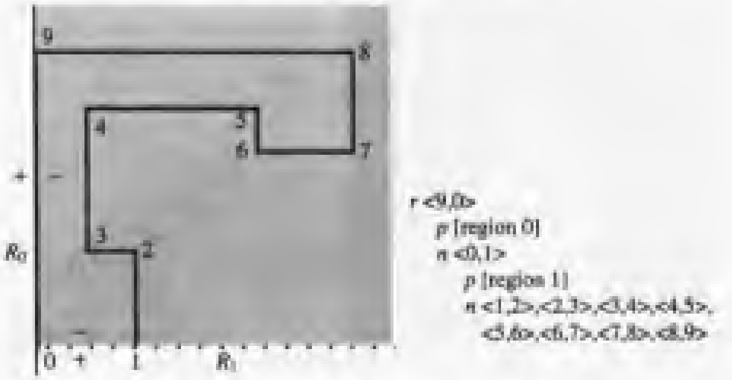


图 13.5　处理边(0, 1)后的当前状态

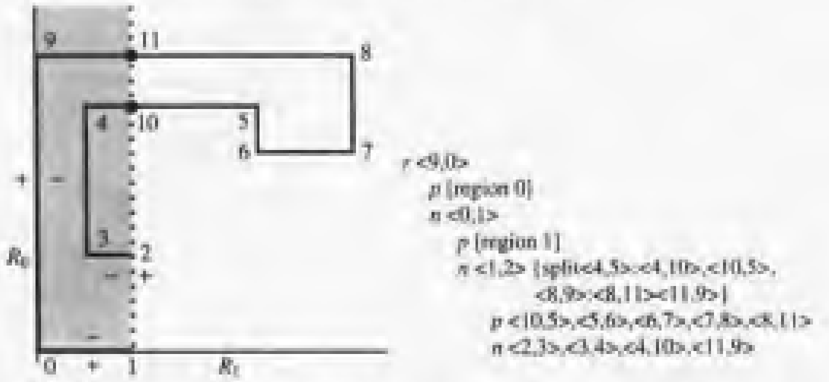


图 13.6　处理边(1, 2)后的当前状态。该边使(4, 5)分裂为(4, 10)和(10, 5)，也使(8, 9)分裂为(8, 11)和(11, 9)

要处理的下一条边是(10, 5)。图 13.7 显示了处理这条边之后的空间分区二叉树的状态。这条边分解边(7, 8)，产生了一个新的顶点，在图中标示为 12。

要处理的下一条边是(5, 6)。图 13.8 显示了处理这条边之后的空间分区二叉树的状态。这一步没有产生新的顶点。

我们将剩余的边留给你来检验，包括(6, 7)，(7, 12)，(12, 8)，(8, 11)，(2, 3)，(3, 4)（迫使边(11, 9)分解，并产生一个标示为 13 的新顶点），(4, 10)，(11, 13)和(13, 9)。空间分区二叉树的最后状态显示在图 13.9 中。图中标示了对应于空间分区二叉树的各个叶子节点的区域。分

区时产生的新顶点在图中显示为黑块。分解(11, 9)时产生的点 13 位于图的最左边。■

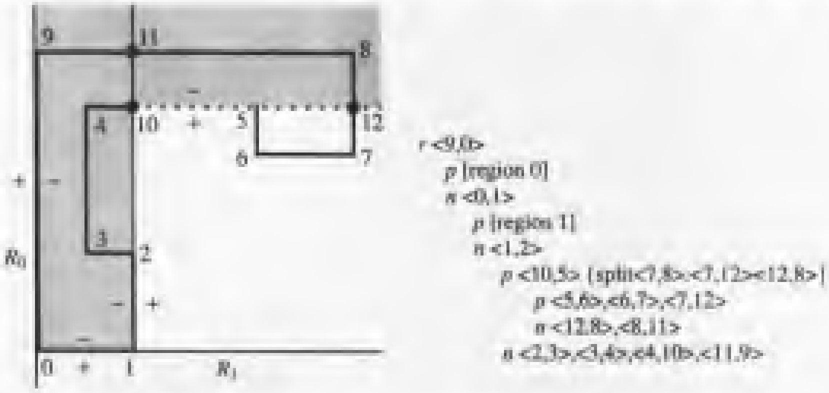


图 13.7　处理边(10, 5)后的当前状态。该边使(7, 8)分裂为(7, 12)和(12, 8)

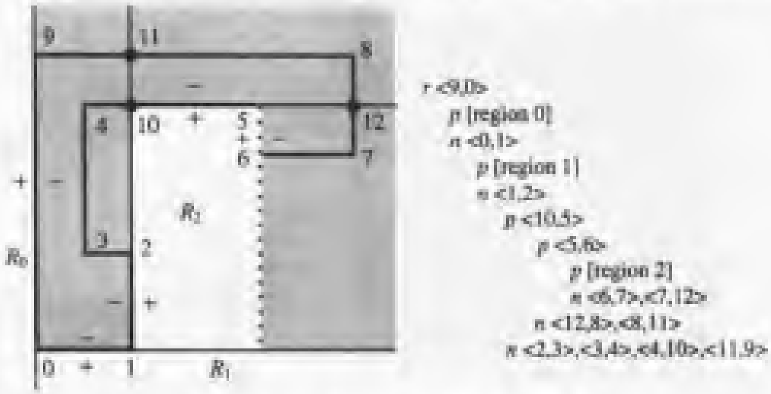


图 13.8　处理边(5, 6)后的当前状态

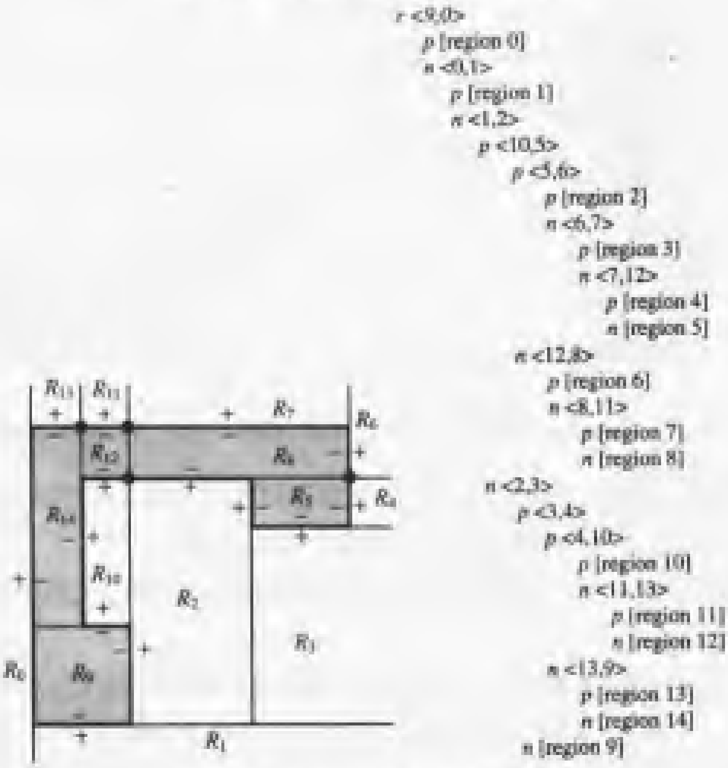


图 13.9　处理边(13, 9)后的最终状态

【**实例**】图 13.10 显示了一个凸多边形的分区及相应的空间分区二叉树。对该凸多边形的二叉树构建要求不分解边。然而，这棵树刚好是节点的一个线性表。在最坏的情形中，任何位于多边形内的包含测试都要求处理树中的每一个节点。可以简化处理的较好的情形是从一棵尽可能平衡的二叉树开始处理。■

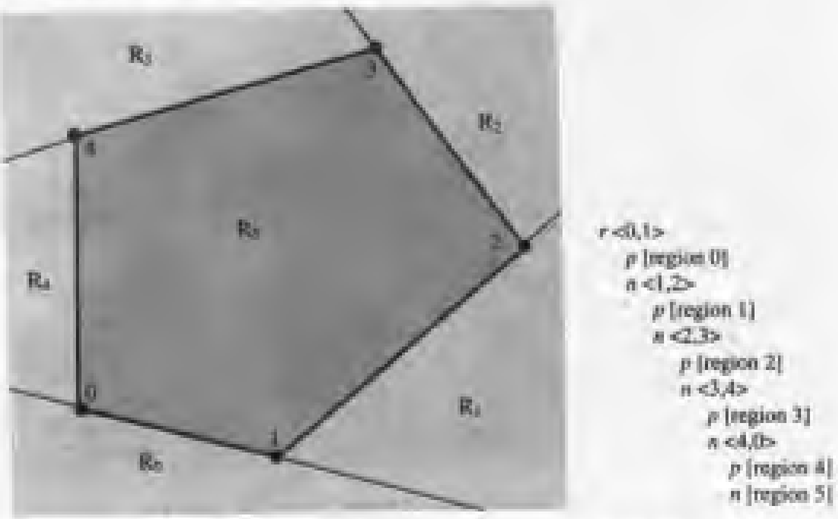


图 13.10　凸多边形的分区及相应的 BSP 树

13.1.2　最小分解与平衡树

正如我们在分析一个凸多边形的空间分区二叉树的例子中所看到的，构建这样的树并不要求进行分解，然而这样的树是非常不平衡的，因为它是一个线性表。构建时的问题是，所选择的分区直线包含多边形的边。对于用多边形来对空间进行分区，这个限制条件并不是必需的。另一种方法是用一种能得到最小分解和平衡树的聪明方法来选择分区直线。对于一个凸多边形，建立这样的树总是可能的。对于一般的多边形，我们并不清楚什么是选择分区直线的最好方法。

对于凸多边形来说，二分法非常有效。这种方法的基本思想是，选择一条直线，该直线包含顶点V_0和另一个将所有的顶点划分为元素个数相同的两个子集的顶点V_m。7.7.2 节中给出了一个计算m的过程，用来寻找凸多边形的极值点：

```
int GetMiddleIndex(int i0, int i1, int N)
{
    if (i0 < i1)
        return (i0 + i1) / 2;
    else
        return (i0 + i1 + N) / 2 (mod N);
}
```

值N是顶点的数量。在初始调用时，将i_0和i_1都置零。条件$i_0 < i_1$将得到一个很显然的结果——返回的索引是输入的顶点的平均值，当然这也与该函数的名称相符。例如，如果多边形具有$N = 5$个顶点，输入$i_0 = 0$ 和$i_1 = 2$ 将得到返回索引为 1。其他的条件处理索引集合。如果$i_0 = 2$且$i_1 = 0$，那么隐含的有序索引集为$\{2, 3, 4, 0\}$。由于$3 = (2 + 0 + 5)/2 \pmod 5$，因此中索引选为 3。

因为分区直线经过顶点，并且多边形是凸多边形，因此这条直线并不分解任何边。由于使用二分法，因此这棵树将被自动地平衡。图 13.11 显示了最后一个例子中凸多边形的分区和空间分区二叉树。注意，这棵树的深度小于用原来的方法构建的树。

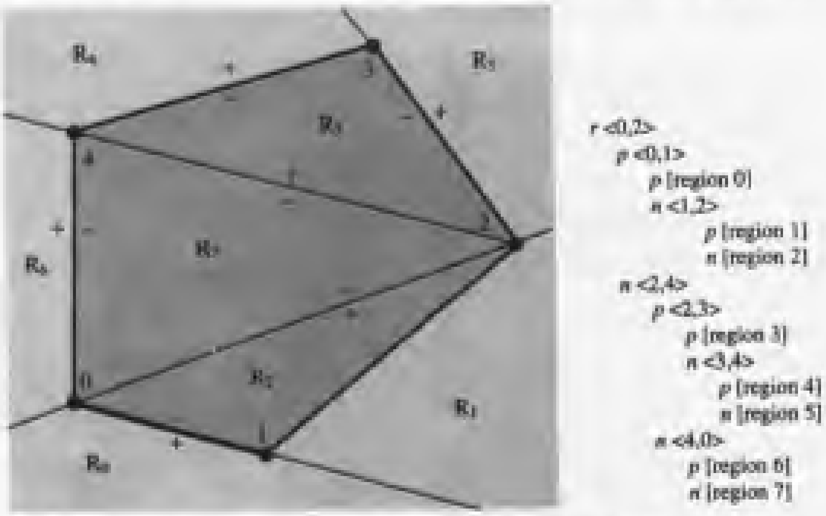


图 13.11　凸多边形的分区及相应的平衡 BSP 树

13.1.3　用空间分区二叉树进行点在多边形内的检测

一个多边形的一种空间分区二叉树表示自然地提供了检测一个点位于多边形内、多边形外还是位于多边形上的能力。在这棵树的每一个节点上处理该点，检测它位于分区直线的哪一侧。如果处理到达了一个叶子节点，那么该点就位于对应的凸区域内。如果该区域位于这个多边形内部（或者外部），那么该点就位于这个多边形内部（或者外部）。在任何一个节点上，如果该点位于分区直线所包含的一条边上，则该点自然位于多边形上。伪码列出如下。如果该点位于多边形外，则返回值为+1；如果该点位于多边形内，则返回值为−1；如果该点位于多边形上，则返回值为 0。

```
int PointLocation(BspTree T, Point P)
{
    // test point against splitting line
    type = Classify(T.line, P);
    if (type is POSITIVE) {
        if (T.posChild exists)
            return PointLocation(T.posChild, P);
        else
            return +1;
    } else if (type is NEGATIVE) {
        if (T.negChild exists)
            return PointLocation(T.negChild, P);
        else
            return -1;
    } else {
        // type is COINCIDENT
        for (each edge T.coincident[i]) {
            if (P on T.coincident[i])
```

```
                return 0;
            }

            // does not matter which subtree you use
            if (T.posChild exists)
                return PointLocation(T.posChild, P);
            else if (T.negChild exists)
                return PointLocation(T.negChild, P);
            else {
                // Theoretically you should not get to this block. Numerical
                // errors might cause the block to be reached, most likely
                // because the test point is nearly an end point of a
                // coincident edge. An implementation could throw an exception
                // or 'assert' in Debug mode, but still return a value in Release
                // mode. For simplicity, let's just return 0 in hopes the test
                // point is nearly a coincident edge end point.
                return 0;
            }
        }
    }
```

13.1.4 用空间分区二叉树分解线段

给定位于一个平面上的一条线段，自然可以用空间分区二叉树来将线段分解为一系列子线段，它们可包含于一个外部区域、包含于一个内部区域或者与分区直线重合。在这棵树的每一个节点上处理这条线段。如果这条线段位于这条直线的正侧，可能有一个端点位于这条直线上，那么就将它送往正子树，以进行进一步的处理。如果这个节点没有正子树，那么这条线段就位于一个外部区域内。类似地，如果这条线段位于这条直线的负侧，就由负子树来对它做进一步的处理，除非这个节点没有负子树，此时这条线段就位于一个内部区域内。如果这条线段穿过分区直线，那么它被分解为两段，一段位于这条直线的正侧，另一段位于这条直线的负侧。位于正侧的段由正子树做进一步的处理；位于负侧的段由正负树做进一步的处理。最后一种可能是，这条线段与分区直线重合。必须计算这条线段与产生分区直线的边的相交情况。不被边所包含的任何子线段都必须由正子树和负子树进行进一步的处理。

进行线段处理所得到的最终结果是线段分解，即将线段表示为相邻的子线段的联合。每一条子线段都位于一个内部区域、一个外部区域或者位于多边形的边界上。位于多边形边界上的线段可以根据多边形的边的方向和线段的方向来进行进一步的分类。当用空间分区二叉树来对多边形进行布尔运算时，这种方法非常重要。图 13.2 显示了一条分区直线和两条线段，E_0和E_1，它们都与这条直线重合。E_0与N的方向相同，但是E_1与N的方向相反。

图 13.12 显示了图 13.3 中多边形和一条与多边形相交的线段。这条线段的端点标记为 0 和 1。当这条线段沿这棵树递归地处理时，其他的标记为 2 至 6 的点被插入分区中。图的右边部分显示了这个多边形的空间分区二叉树。线段表示为(i_0, i_1)。开始时，在这棵树的根节点上处理线段(0, 1)。这条线段没有被包含边(9, 0)的直线分解，因此区域 0 不包含原始线段的任何部分。线段(0, 1)被送往根节点的父子树处理。这条线段被包含边(0, 1)的直线分

解，新的点标示为 2。线段(0, 2)位于分区直线的正侧，然而根节点没有正子树，因此线段(0, 2)包含于区域 1。线段(2, 1)被送往负子树，并重复这种处理。

最后的分区得到正线段(0, 2), (3, 5)和(6, 1)。负线段为(2, 3), (5, 4)和(4, 6)。注意，并不要求子线段正负交替出现。在上一个例子中，子线段(5, 4)和(4, 6)是相邻的，然而它们都是负的。线段分解的实现能够处理这类情形，并将相邻的具有相同符号的线段合并为一条线段。

下面列出了进行线段分解的伪码。输入参数为树、多边形，以及线段端点V_0 和V_1。输出是子线段的 4 个集合。Pos集合包含位于正区域的子线段，而Neg 集合包含位于负区域的子线段。其余的两个集合存储包含于与分区直线重合的边的子线段。CoSame 集合存放包含于每一个子线段都与边同向的边的子线段。CoDiff集合存放包含于每一个子线段都与边反向的边的子线段。

```
void GetPartition(BspTree T, Edge E, EdgeSet Pos, EdgeSet Neg,
    EdgeSet CoSame, EdgeSet CoDiff)
{
    type = Classify(T.line, E, SubPos, SubNeg);
    if (type is CROSSES) {
        GetPosPartition(T.posChild, SubPos, Pos, Neg, CoSame, CoDiff);
        GetNegPartition(T.negChild, SubNeg, Pos, Neg, CoSame, CoDiff);
    } else if (type is POSITIVE) {
        GetPosPartition(T.posChild, E, Pos, Neg, CoSame, CoDiff);
    } else if (type is NEGATIVE) {
        GetNegPartition(T.negChild, E, Pos, Neg, CoSame, CoDiff);
    } else {
        // type is COINCIDENT
        // construct segments of E intersecting coincident edges
        A = {E};
        for (each edge E' in T.coincident)
            A = Intersection(A, E');

        for (each segment S of A) {
            if (S is in the same direction as T.line)
                CoPos.Insert(S);
            else
                CoNeg.Insert(S);
        }

        // construct segments of E not intersecting coincident edges
        B = {E} - A;
        for (each segment S of B) {
            GetPosPartition(T.posChild, S, Pos, Neg, CoSame, CoDiff);
            GetNegPartition(T.negChild, S, Pos, Neg, CoSame, CoDiff);
        }
    }
}

void GetPosPartition(BspTree T, Edge E, EdgeSet Pos, EdgeSet Neg,
    EdgeSet CoSame, EdgeSet CoDiff)
{
```

```
    if (T.posChild)
        GetPartition(T.posChild, E, Pos, Neg, CoSame, CoDiff);
    else
        Pos.Insert(E);
}

void GetNegPartition(BspTree T, Edge E, EdgeSet Pos, EdgeSet Neg,
    EdgeSet CoSame, EdgeSet CoDiff)
{
    if (T.negChild)
        GetPartition(T.negChild, E, Pos, Neg, CoSame, CoDiff);
    else
        Neg.Insert(E);
}
```

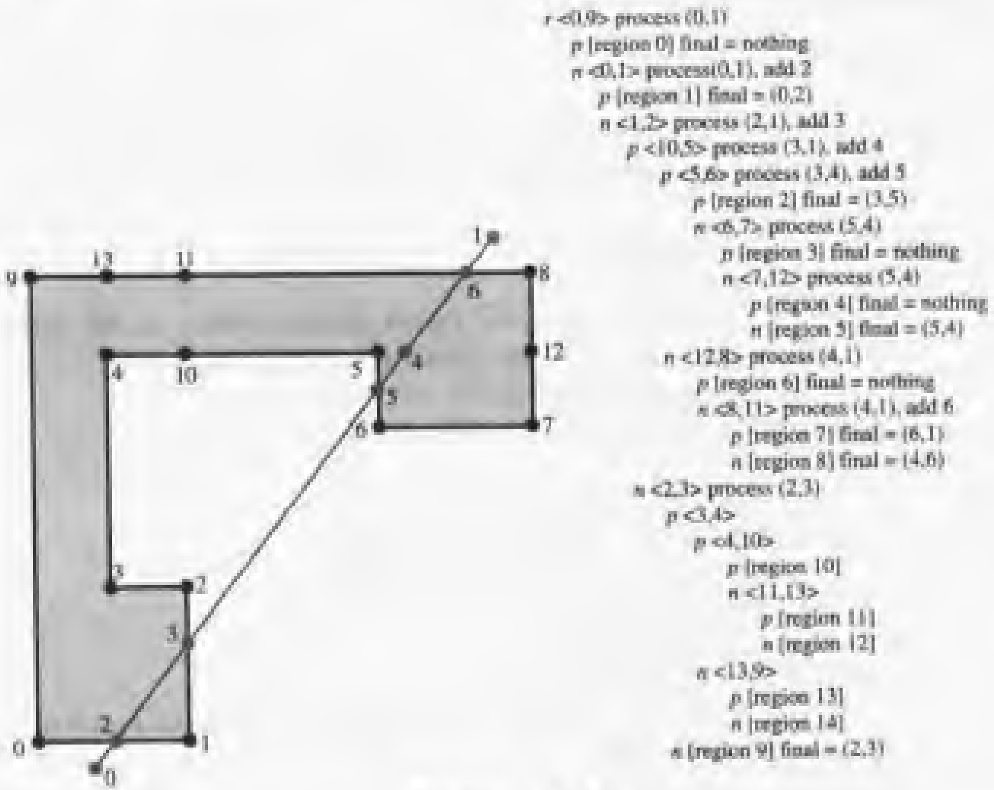


图 13.12　线段分区

13.2　三维空间分区二叉树

我们建议读者在阅读本节之前先阅读 13.1 节以获得关于空间分区二叉树的有关知识。

考虑一个方程为 $\vec{n}\cdot X-c=0$的空间上的平面。这个平面将空间分区为两个"半空间"。位于$\vec{n}$所指向的平面一侧的半空间叫做平面的正侧；另一侧叫做平面的负侧。如果X位于正侧，那么$\vec{n}\cdot X-c>0$，因此使用术语"正侧"来表示。如果X位于负侧，那么$\vec{n}\cdot X-c<0$。如果X位于平面上，自然有 $\vec{n}\cdot X-c=0$。

每一个半空间都可以用空间上的另外的平面来进行分区。得到的正的和负的区域又可以被进一步分区，用二叉树来表示得到的分区结果，二叉树的每一个节点都表示一个分区

平面。节点的左子树表示该节点所表示的分区平面的正侧；节点的右子树表示该节点所表示的分区直线的负侧。二叉树的叶节点表示分区得到的凸区域。

13.2.1 多面体的空间分区二叉树表示

正如二维空间分区二叉树可用来将多边形分区成凸子多边形一样，三维空间分区二叉树可用来将多面体分区成凸子多面体。这种分解对点在多面体内的检测和对多面体的布尔运算都非常有用。最简单的构建方法是利用多边形面来确定分区平面。与二维空间的情形一样，应用递归分区，惟一的复杂之处是计算一个凸多面体与一个平面相交比计算一条线段与一条直线相交稍微困难一些。下面列出了由一个多面体来构建一棵空间分区二叉树的伪码。提醒一下，我们要求多面体的面都是凸多边形。

```
BspTree ConstructTree(FaceList L)
{
    T = new BspTree;

    // use a face to determine the splitting plane for the tree node
    T.plane = GetPlaneFromFace(L.first);  // Dot(N, X) - c = 0

    FaceList posList, negList;  // initially empty lists
    for (each face F of L) {
        // Determine how face and plane relate to each other.  If the face
        // crosses the plane, the subpolyhedra on the positive and
        // negative side of the plane are returned.
        type = Classify(T.plane, F, SubPos, SubNeg);

        if (type is CROSSES)  {
            // Dot(N, X) - c < 0 for some vertices,
            // Dot(N, X) - c > 0 for some vertices
            posList.AddFace(SubPos);
            negList.AddFace(SubNeg);
        } else if (type is POSITIVE) {
            // Dot(N, X) - c >= 0 for all vertices, at least one positive
            posList.AddFace(F);
        } else if (type is NEGATIVE) {
            // Dot(N, X) - c <= 0 for all vertices, at least one negative
            negList.AddFace(F);
        } else {
            // type is COINCIDENT
            // Dot(N, X) - c = 0 for all vertices
            T.coincident.AddFace(F);
        }
    }

    if (posList is not empty)
        T.posChild = ConstructTree(posList);
    else
        T.posChild = null;
```

```
        if (negList is not empty)
            T.negChild = ConstructTree(negList);
        else
            T.negChild = null;

        return T;
    }
```

函数GetPlaneFromFace产生一个平面，其法线向量指向位于指定的面的多面体的外部区域。其他的一致面的法线方向与这个平面的法线方向并不一定相同。这与图 13.2 中显示的二维空间的情形相似。

函数Classify用于寻找当前的面与节点平面的相交线段。如果存在一个相交线段为这个面的内部线段，就返回正的和负的子面。如果这个面只是位于正侧，并且一些顶点或者一致的边可能位于这个平面上，那么这个面可归类为正面。对于面位于这个平面的负侧的情形，可以做类似的分类。这个面也可以完全位于分区平面上，在这种情形中，这个面归类为一致面。该函数的伪码如下

```
int Classify(Plane P, Face F, Face SubPos, Face SubNeg)
{
    for (i = 0; i < F.vertexQuantity; i++)
        d[i] = Dot(P.normal, F.V(i) - P.origin);

    if (at least one d[i] > 0 and at least one d[i] < 0) {
        // face crosses plane
        SplitPolygon(F, P, d[], SubPos, SubNeg);
        return CROSSES;
    } else if (all d[i] >= 0 with at least one d[i] > 0)  {
        // All vertices of the face are on the positive side of the plane,
        // but not all vertices are on the plane.
        return POSITIVE;
    } else if (all d[i] <= 0 with at least one d[i] < 0) {
        // All vertices of the face are on the negative side of the plane,
        // but not all vertices are on the plane.
        return NEGATIVE;
    } else {
        // All vertices of the face are on the plane.
        return COINCIDENT;
    }
}
```

函数SplitPolygon确定与平面相交的F的边（可通过d[]的值来计算），并构建两个子多边形。计算边与平面的交点时，可以使用类似于处理二维空间中计算边与直线之间的交点的方法。当一个或多个d[i]接近于零时，实现代码必须处理浮点舍入误差。

13.2.2 最小分解与平衡树

对于二维空间中的问题，由一个凸多边形可以得到一个线性表的空间分区二叉树。一个凸多面体也具有线性表的空间分区二叉树，这是因为多面体的面总是位于一个面所在平

面的负侧。在二维空间中，通过选择连接两个不相邻顶点的直线作为分区直线，就可以构建一棵平衡树。多边形为凸的性质保证分区直线不会分解多边形的任何边。得到的树是非常理想的树：它是平衡的，并且不要求分解边。

三维空间的情形并没有如此简单。可以选取一个将凸多面体分成两部分的分区平面，但是在大多数情形中，这个平面将要求对面进行分解。惟一一种不会出现分解的情形是当这个平面与多面体仅相交于一条边时的情形。这意味着多面体必须包含一条共面的由边组成的折线——这并不是一般的情形。其次，获得一棵平衡树将要求进行一些分解，我们希望分解的次数尽可能地少。选择一种最好的可以得到最小分解的一般算法的启发式方法是非常困难的。如果实现代码使用关于问题的数据集的具体性质，就更有可能得到好的结果。在任何情形中，用于建立树的算法的质量都是决定空间分区二叉树系统的性能的关键。

13.2.3 用空间分区二叉树进行点在多面体内的检测

计算点相对于多面体的位置的算法，与处理二维空间中的同类问题的算法完全相同。将点与空间分区二叉树的每一个节点进行比较。如果这个点位于平面的正侧，并且正子树存在，那么该点由这棵子树进行进一步的处理。如果正子树不存在，那么它位于多面体外。如果该点位于平面的负侧，并且负子树存在，那么该点由这棵子树进行进一步的处理。如果负子树不存在，那么它位于多面体内。如果该点位于分区平面上，那么它包含于一个面内，此时点位于多面体上，或者该点不包含于一个面内，将它送往任何存在的子树做进一步处理。伪码列出如下。如果该点位于多面体外，则返回值为+1，如果该点位于多面体内，则返回值为−1，或者如果该点位于多面体上，则返回值为 0。

```
int PointLocation(BspTree T, Point P)
{
    // test point against splitting plane
    type = Classify(T.plane, P);
    if (type is POSITIVE) {
        if (T.posChild exists)
            return PointLocation(T.posChild, P);
        else
            return +1;
    } else if (type is NEGATIVE) {
        if (T.negChild exists)
            return PointLocation(T.negChild, P);
        else
            return -1;
    } else {
        // type is COINCIDENT
        for (each face T.coincident[i]) {
            if (P on T.coincident[i])
                return 0;
        }

        // does not matter which subtree you use
        if (T.posChild exists)
            return PointLocation(T.posChild, P);
```

```
        else if (T.negChild exists)
            return PointLocation(T.negChild, P);
        else
            return 0;
    }
}
```

13.2.4 用空间分区二叉树分解线段

给定空间的一条线段，自然可用空间分区二叉树来将线段分解为一系列的子线段，它们可包含于一个外部区域、包含于一个内部区域或者与分区平面重合。在这棵树的每一个节点上处理这条线段。如果这条线段位于这个平面的正侧，可能有一个端点位于这个平面上，那么就将它送往正子树，以进行进一步的处理。如果这个节点没有正子树，那么这条线段就位于一个外部区域内。类似地，如果这条线段位于这个平面的负侧，就由负子树来对它做进一步的处理，除非这个节点没有负子树，此时这条线段就位于一个内部区域内。如果这条线段穿过分区平面，那么它被分解为两段，一段位于这个平面的正侧，另一条位于这个平面的负侧。位于正侧的段由正子树做进一步的处理；位于负侧的段由正负树做进一步的处理。最后一种可能是，这条线段与分区平面重合。必须计算这条线段与产生分区平面的面的交集。注意，计算一条线段与一个二维多边形的交集确实是一个问题，可以利用二维空间分区二叉树来解决这个问题。然而，由于要求面是凸多边形，因此可以用非常直接的方法来实现线段与凸多边形的交集，并不需要构建面的空间分区二叉树。不被面所包含的任何子线段都必须由正子树和负子树进行进一步的处理。

进行线段处理所得到的最终结果是线段分解，即将线段表示为相邻的子线段的联合。每一个子线段都位于一个内部区域、一个外部区域或者位于多边形的边界上。下面列出了线段分解的伪码。

```
void GetPartition(BspTree T, Edge E, EdgeSet Pos, EdgeSet Neg, EdgeSet Coin)
{
    type = Classify(T.plane, E, SubPos, SubNeg);
    if (type is CROSSES) {
        GetPosPartition(T.posChild, SubPos, Pos, Neg, Coin);
        GetNegPartition(T.negChild, SubNeg, Pos, Neg, Coin);
    } else if (type is POSITIVE) {
        GetPosPartition(T.posChild, E, Pos, Neg, Coin);
    } else if (type is NEGATIVE) {
        GetNegPartition(T.negChild, E, Pos, Neg, Coin);
    } else {
        // type is COINCIDENT
        // construct segments of E intersecting coincident faces
        A = {E};
        for (each face F in T.coincident)
            A = Intersection(A, F);
         for (each segment S of A)
             Coin.Insert(S);

         // construct segments of E not intersecting coincident faces
```

```
            B = {E} - A;
            for (each segment S of B) {
                GetPosPartition(T.posChild, S, Pos, Neg, Coin);
                GetNegPartition(T.negChild, S, Pos, Neg, Coin);
            }
        }
    }

    void GetPosPartition(BspTree T, Edge E, EdgeSet Pos, EdgeSet Neg, EdgeSet Coin)
    {
        if (T.posChild)
            GetPartition(T.posChild, E, Pos, Neg, Coin);
        else
            Pos.Insert(E);
    }

    void GetNegPartition(BspTree T, Edge E, EdgeSet Pos, EdgeSet Neg, EdgeSet Coin)
    {
        if (T.negChild)
            GetPartition(T.negChild, E, Pos, Neg, Coin);
        else
            Neg.Insert(E);
    }
```

用于空间分区二叉树的函数Classify分解一个面，然而用于该过程中的函数Classify要做的工作更简单，即只需分解一条边。

线段并不一定必须表示为在某一给定时间由连续的点所构成的几何实体。例如，在一个碰撞检测系统中，用点来抽象地表示位于被一棵空间分区二叉树所分区的空间中的运动物体，用线段来表示点在一个指定的时间区间内运动的预测路径。如果点所表示的物体不允许穿过分离内部区域和外部区域的（包含于空间分区二叉树的分区平面）“墙”，那么分区线段可用于阻止物体这样做。如果线段被分区平面所分解，那么包含（处于零时间的）初始点的最短子线段就表示物体可以移动多远而不发生碰撞。应用程序因此可以将物体移动这样的距离，或者，如果愿意的话，可以避免任何碰撞。这种处理碰撞的方法优于下面的方法：对投影路径采样以得到一系列点，然后用空间分区二叉树来处理每一个点，以确定它是否包含于一个内部或者外部区域，然后集中分析这些结果，以确定物体可以运动多远。

13.2.5 用空间分区二叉树分解凸多边形

用三维空间分区二叉树来分解凸多边形的方法与用二维空间分区二叉树来分解线段的方法完全类似。伪码列出如下。假设输入的面F为凸多边形。算法中最复杂的部分是处理当多边形与分区平面重合时的情形的一段代码，在这种情形中，问题可以简化为计算多边形的相交和差。13.5节显示了如何计算多边形的相交和差。

```
void GetPartition(BspTree T, Face F, FaceSet Pos, FaceSet Neg,
    FaceSet CoPos, FaceSet CoNeg)
{
    type = Classify(T.plane, F, SubPos, SubNeg);
    if (type is CROSSES) {
```

```
        GetPosPartition(T.posChild, SubPos, Pos, Neg, CoPos, CoNeg);
        GetNegPartition(T.negChild, SubNeg, Pos, Neg, CoPos, CoNeg);
    } else if (type is POSITIVE) {
        GetPosPartition(T.posChild, F, Pos, Neg, Coin);
    } else if (type is NEGATIVE) {
        GetNegPartition(T.negChild, F, Pos, Neg, Coin);
    } else {
        // type is COINCIDENT
        // compute intersection of F with coincident faces
        A = {F};
        for (each face F' in T.coincident)
            A = Intersection(A, F');

        for (each face S of A) {
            if (S has normal in same direction as T.plane)
                CoPos.Insert(S);
            else
                CoNeg.Insert(S);
        }

        // construct complement of intersection of F with coincident faces
        B = {F} - A;
        for (each face S of B) {
            GetPosPartition(T.posChild, S, Pos, Neg, CoPos, CoNeg);
            GetNegPartition(T.negChild, S, Pos, Neg, CoPos, CoNeg);
        }
    }
}

void GetPosPartition(BspTree T, Face F, FaceSet Pos, FaceSet Neg,
    FaceSet CoPos, FaceSet CoNeg)
{
    if (T.posChild)
        GetPartition(T.posChild, F, Pos, Neg, CoPos, CoNeg);
    else
        Pos.Insert(F);
}

void GetNegPartition(BspTree T, Face F, FaceSet Pos, FaceSet Neg,
    FaceSet CoPos, FaceSet CoNeg)
{
    if (T.negChild)
        GetPartition(T.negChild, F, Pos, Neg, CoPos, CoNeg);
    else
        Neg.Insert(F);
}
```

13.3　点在多边形内的检测

在图形学应用程序中，确定一个点是否位于一个多边形内是一种常见的问题。在

Heckbert（1994）的“检测点在多边形内的策略”一节中提供了关于这类方法的一个调查。如果该多边形表示为空间分区二叉树，那么 13.1 节讨论了确定一个点位于多边形的内部还是外部的方法。我们在此讨论一些不需要进行预处理（不需要建立支持快速处理的数据结构）的方法。最后一节讨论了一种需要预处理的方法，其中的预处理是将多边形分解为一些梯形。

13.3.1　点在三角形内的检测

考虑一个点P和一个具有不共线的顶点V_i（$0 \leqslant i \leqslant 2$）的三角形。设三角形的边分别为$\vec{e}_0 = V_1 - V_0, \vec{e}_1 = V_2 - V_1$和$\vec{e}_2 = V_0 - V_2$。边的法线为$\vec{n}_i = \mathrm{Perp}(\vec{e}_i)$，其中$\mathrm{Perp}(x, y) = (y, -x)$。如果顶点是按逆时针排列的，则法线指向外；如果顶点是顺逆时针排列的，则法线指向里。

在顶点是逆时针排列的情形中，如果P位于每一条边所在的直线$\vec{n}_i \cdot (X - V_i) = 0$的负侧，则它位于三角形内。即当$\vec{n}_i \cdot (P - V_i) < 0$对所有的$i$都成立时，$P$位于三角形内。当$\vec{n}_i \cdot (P - V_i) > 0$对至少一个$i$成立时，$P$位于三角形外。$P$也可能位于三角形的边界上，此时$\vec{n}_i \cdot (P - V_i) \leqslant 0$对所有$i$成立，并且至少有一个$i$使得等式成立。如果等式出现一次，那么点位于一条边上，但并不位于一个顶点上。如果等式出现两次，那么该点是一个顶点。等式不可能出现三次。如果顶点是顺时针排列的，那么只需反转上述测试中的不等式。在所有三角形的顶点排列方向不一致的应用程序中，与顶点排序无关的检测点在三角形内的不等式为

$$(\vec{n}_0 \cdot (V_2 - V_0))(\vec{n}_i \cdot (P - V_i)) > 0 \text{ 对所有的} i \text{成立}$$

当然，第一个点积有效地确定了顶点的排序。

这个点也可以写成用中心坐标来表示的形式，即$P = c_0V_0 + c_1V_1 + c_2V_2$，其中$c_0 + c_1 + c_2 = 1$。如果$0 \leqslant c_i \leqslant 1$对所有$i$都成立，则$P$位于三角形内或者位于三角形上。如果$c_j < 0$对至少一个$j$成立，那么该点位于三角形外。系数$c_2$可用如下的方法来计算：

$$\begin{aligned} P - V_0 &= (c_0 - 1)V_0 + c_1V_1 + c_2V_2 = (-c_1 - c_2)V_0 + c_1V_1 + c_2V_2 \\ &= c_1(V_1 - V_0) + c_2(V_2 - V_0) \end{aligned}$$

因此$\vec{n}_0 \cdot (P - V_0) = c_2\vec{n}_0 \cdot (V_2 - V_0)$。可对$c_0$和$c_1$应用类似的方法，以得到

$$c_0 = -\frac{\vec{n}_1 \cdot (P - V_1)}{\vec{n}_1 \cdot \vec{e}_0}, \quad c_1 = -\frac{\vec{n}_2 \cdot (P - V_2)}{\vec{n}_2 \cdot \vec{e}_1}, \quad c_2 = -\frac{\vec{n}_0 \cdot (P - V_0)}{\vec{n}_0 \cdot \vec{e}_2}$$

如果点P位于三角形内，那么对于检测来说，它的精确表示并不重要。对于给定的三角形排序，上述分数的分母都具有相同的符号，因此分子的符号完全确定了问题的解。它们的符号与在本节前面提供的检测中的讨论完全一样。

虽然我们假设三角形的顶点是不共线的，但应用程序可能还是需要处理顶点共线的退化情形。更可能的情形是，在一个数据集中可能遇到三角形是针状的或者具有很小的面积的情形。这样的三角形满足不共线的条件，但是浮点数的舍入误差可能会引起问题。

考虑三个不同的顶点共线的情形，此时三角形退化为一条线段。其中必定有一个顶点是其他两个顶点的连线的内点。为了便于讨论，假设V_2为这个顶点。它们的方向向量都互相平行，但是$\vec{n}_0$指向与$\vec{n}_1$和$\vec{n}_2$相反的方向。如果P不在顶点的连线上，那么$\mathrm{Sign}(\vec{n}_0 \cdot (P -$

$V_0)) = -\mathrm{Sign}(\vec{n}_1 \cdot (P - V_1))$和$\mathrm{Sign}(\vec{n}_1 \cdot (P - V_1)) = \mathrm{Sign}(\vec{n}_2 \cdot (P - V_2))$。所有符号都相同是不可能的，因此前面提到的点在三角形内的符号检测仍然可以得到正确的结果，即P位于三角形之外（在这种情形中，就是不在线段上）。如果P位于这些顶点的连线上，那么所有的三个符号都为零——没有足够的信息可以确定P是否包含于线段内。必须进行进一步的工作才能解决这个问题。特别地，P必须是线段的端点的线性组合，即$P = (1-t)V_0 + tV_1$对某些t值成立。如果$t \in [0,1]$，那么P包含于线段内，因而P位于三角形"内"。否则，它位于三角形之外。这种类型的分析也可以用于二维空间中一个点集的凸包的构建（13.7 节）。

如果三角形是针状的，就是说几乎是共线的，那么浮点数舍入误差可能使这种情形看起来像是共线的情形。如果两个顶点几乎是共线的，也可能引起同样的问题。第一种方法是对三角形进行预处理，以将几乎退化的三角形变成线段或者点，然后分别利用点是否相等、点在线段上或者点在三角形内的检测方法。如果对同一组三角形需要进行大量的包含测试，那么我们推荐使用这种方法。第二种方法将这类三角形看成是三角形，并进行退化检测，再转化为需要进行点是否相等或者点在线段上的检测。

13.3.2　点在凸多边形内的检测

进行点在三角形内的检测所使用的点位于线的哪一侧的检测方法可以自然地扩展，用于确定一个点是否位于一个凸多边形内。设凸多边形具有逆时针排序、不共线的顶点$\{V_i\}_{i=0}^{n-1}$，其中$V_n = V_0$。如果顶点都共线，或者多边形几乎退化，成为针状，那么需要处理在点在三角形内的检测中所描述的数值问题。

多边形的边为$\vec{e}_i = V_{i+1} - V_i$，并且指向外面的边法线为$\vec{n}_i = \mathrm{Perp}(\vec{e}_i)$。当$\vec{n}_i \cdot (P - V_i) < 0$对所有的$i$都成立时，点$P$位于多边形内。如果$\vec{n}_i \cdot (P - V_i) > 0$对某些$i$成立，则该点位于多边形外。如果$\vec{n}_i \cdot (P - V_i) \leqslant 0$对所有的$i$都成立，而且至少有一个$i$使等式成立，那么该点位于多边形的边界上。如果等式仅对一个i成立，那么P位于一条边上，但不是顶点。如果等式对两个i成立，那么该点是一个顶点。等式不能出现三次或者三次以上。

该算法的级别是$O(n)$，因为为了检测点P是否位于多边形内，必须检测多边形的所有n条边。检测一个点是否位于多边形内的简单实现为

```
bool PointInPolygon(Point P, ConvexPolygon C)
{
    for (i0 = 0, i1 = C.N - 1; i < C.N; i0++) {
        if (Dot(Perp(C.V(i0) - C.V(i1)), P - C.V(i1)) > 0)
            return false;
    }
    return true;
}
```

当P位于多边形内时，循环体需要执行n次。

1．一种渐近较快的方法

另一种算法利用二分法，我们也使用二分法来构建凸多边形的平衡空间分区二叉树。为了说明这种方法，考虑一个具有逆时针排序的顶点V_i（$0 \leqslant i \leqslant 3$）的四边形。该多边形

可看做两个三角形$\langle V_0, V_1, V_2\rangle$和$\langle V_0, V_2, V_3\rangle$的联合。二分法的实现为

```
bool PointInConvexQuadrilateral(Point P, ConvexPolygon C)
{
    if (Dot(Perp(C.V(2) - C.V(0)), P - C.V(0)) > 0) {
        // P potentially in <V0, V1, V2>
        if (Dot(Perp(C.V(1) - C.V(0)), P - C.V(1)) > 0) return false;
        if (Dot(Perp(C.V(2) - C.V(1)), P - C.V(1)) > 0) return false;
    } else {
        // P potentially in <V0, V2, V3>
        if (Dot(Perp(C.V(3) - C.V(2)), P - C.V(3)) > 0) return false;
        if (Dot(Perp(C.V(0) - C.V(3)), P - C.V(3)) > 0) return false;
    }
    return true;
}
```

当P位于多边形内时，需要计算3个点积。简单实现需要计算4个点积。然而，简单实现用一次点积来标记某些位于外面的点，但是处理四边形的二分法至少要进行两次点积才能排除一个位于外面的点。对于一般凸多边形，二分法的实现为

```
int GetMiddleIndex(int i0, int i1, int N)
{
    if (i0 < i1)
        return (i0 + i1) / 2;
    else
        return (i0 + i1 + N) / 2 (mod N);
}

bool PointInSubpolygon(Point P, ConvexPolygon C, int i0, int i1)
{
    if (i1 - i0 is 1 modulo C.N)
        return Dot(Perp(C.V(i1) - C.V(i0)), P - C.V(i0)) <= 0;

    mid = GetMiddleIndex(i0, i1);
    if (Dot(Perp(C.V(mid) - C.V(i0)), P - C.V(i0)) > 0) {
        // P potentially in <V(i0), V(i0 + 1), ... ,V(mid - 1), V(mid)>
        return PointInSubpolygon(P, C, i0, mid);
    } else {
        // P potentially in <V(mid), V(mid + 1), ... ,V(i1 - 1), V(i1)>
        return PointInSubpolygon(P, C, mid, i1);
    }
}

bool PointInPolygon(Point P, ConvexPolygon C)
{
    return PointInSubpolygon(P, C, 0, 0);
}
```

顶点的索引通过与C.N的模来计算。由于使用了二分法，因此对于具有n个顶点的凸多边形，这种算法的级别为$O(\log n)$。

2. 另一种渐近较快的方法

这一节中描述的方法也要求执行$O(\log n)$次点在凸多边形内的检测。

假设多边形的顶点V_i（$0\leq i\leq n$）按逆时针次序存放。计算x方向的极端顶点。使用7.7.2 节中讨论的二分法可在时间$O(\log n)$内实现这点。x方向的最小顶点具有索引i_{min}，而x方向的最大顶点具有索引i_{max}。注意，$i_{min} < i_{max}$并不是必需的。由于顶点是按逆时针方向从$V_{i_{min}}$到$V_{i_{max}}$遍历的，对应的边的外法线的y分量为负或者零，因此如果多边形在极端顶点中的一个上具有垂直边，那么后一种情形至多出现两次。对应的顶点和边被称为多边形的下半部分。类似地，如果顶点是按逆时针方向从$V_{i_{max}}$到$V_{i_{min}}$遍历的，那么对应的边的外法线的y分量为正或者零。对应的顶点和边被称为多边形的上半部分。

设P为要检测包含性的点。使用索引二分法来确定多边形的上半部分的哪一个顶点V_t具有大于或等于P的x值的最小的x值。如果上半部分有垂直边，初始找到的极值索引可以适当地增加或者减小，以从该检测中排除这些边而不影响算法的正确性。类似地，索引二分法可以用于确定多边形的下半部分的哪一个顶点V_b具有小于或等于P的x值的最大的x值。两种二分法的时间级别都是$O(\log n)$。

包含P的垂直直线与初始点为V_t和V_b的两个方向的边相交。如果P位于两条边之间，那么它位于多边形内。否则，它位于多边形之外。图 13.13 说明了本节中的不同概念。

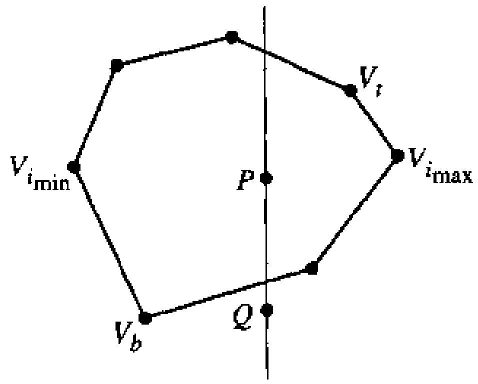


图 13.13 用通过测试点的垂直线来确定与该垂直线相交的两条边，以此进行点在凸多边形内的测试。P在多边形内。Q在多边形外

13.3.3 点在一般多边形内的检测

确定点P是否位于一般多边形内的最常用和最高效的算法，可能都与分析多边形与原点为P且方向为$(1,0)$的射线的交集有关。这种算法的思想与用空间分区二叉树来分解线段的思想完全一样。当射线从P开始遍历时，每当一条边横切穿过时，就进行一次从内到外的转换或者相反的转换。实现有一种应该记录跟踪穿过的次数的奇偶性。奇数次说明P位于多边形内，偶数次说明它位于多边形之外。图 13.14 说明了这种方法。

为了说明其中的一些技术困难，该图中包括了与P_1的射线重合的多边形的边，以及位于这条射线上的多边形的顶点。在多边形的顶点 10 处的问题是，这条射线横切地与多边形的边界相交于该顶点，因此相交应该算做一次穿过。然而，共享该顶点的两条边是分别处理的，每一条边都指出在顶点上的穿过是横切的。其结果是，该顶点被计算了两次穿过，

错误地反转了当前相交计算的奇偶性。顶点 2 的问题稍微不同。射线位于多边形内，分别位于该顶点的左边和右边。应该忽略在顶点 2 处的穿过，因为射线并不横切穿过多边形的边界。对边的分别处理得到了正确的结果，因为两条边都报告了一次射线在顶点 2 的横切穿过。

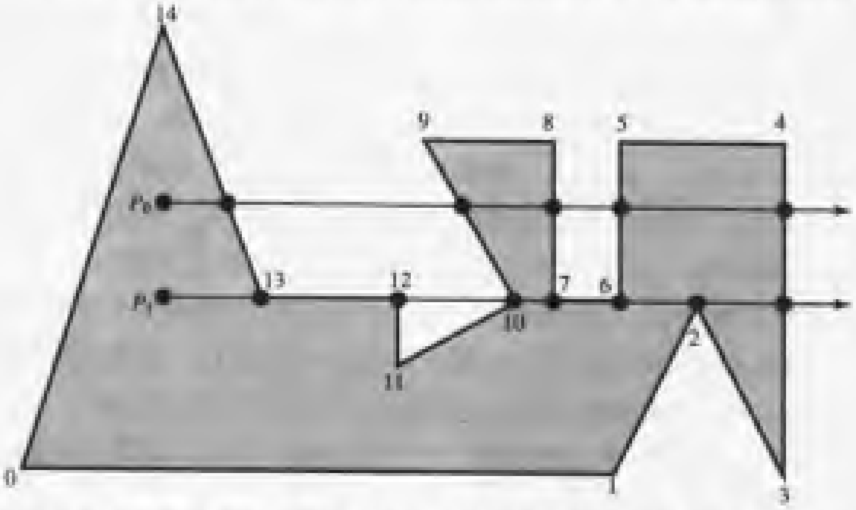


图 13.14　用计算射线与多边形的交点的方法来进行点在多边形内的测试。通过点P_0的射线仅横切通过多边形的边。交点个数是奇数（5），因此该点在多边形内。通过点P_1的射线分析起来要复杂得多

重合的多边形的边 ⟨12, 13⟩ 的问题是，当沿着连接边 ⟨11, 12⟩ 和 ⟨12, 13⟩ 的边水平地观察时，它表现为像是一个顶点。如果顶点 13 重新定位为顶点 12，内 / 外计数并不会改变——在顶点 12 处的穿过是横切多边形的边界的。乍看它表现为似乎可以忽略这条边，作为横切穿过，然而，这是不对的。考虑重合边⟨6, 7⟩。如果顶点 7 重新定位于顶点 6，其中的v形连接就如顶点 2 的计算一样进行计数，因此它并不是一次横切穿过。

Preparata 和 Shamos（1985）提出了如何处理这些构形。Haines（1989）和 O'Rourke（1998）也提出了相同的方法。如果多边形一条边的一个端点严格位于射线的上方，而另一个端点严格地位于射线的下方，则这条边可被看做与射线横切相交。根据这一原则，重合边不能被看成是横切相交的，因此可以被忽略。如果两条边位于射线之上，且它们的共享顶点位于射线上，则它们都可被看成是横切边。如果两条边位于射线之下，且它们的共享顶点位于射线上，则它们都不被看成是横切边。如果一条边位于射线之上，另一条边位于射线之下，且它们的共享顶点位于射线上，则位于射线之上的被看做横切边，而位于射线之下的则不被看做横切边。这一算法的伪码列出如下。

```
bool PointInPolygon(Point P, Polygon G)
{
    bool inside = false;
    for (i = 0, j = G.N - 1; i < G.N; j = i, i++)  {
        U0 = G.V(i);  U1 = G.V(j);

        if ((U0.y <= P.y and P.y < U1.y)  // U1 is above ray, U0 is on or below ray
            or
           (U1.y <= P.y and P.y < U0.y)) // U0 is above ray, U1 is on or below ray
        {
            // Find x-intersection of edge with ray.  Only consider edge
```

```
                // crossings on the ray to the right of P.
                x = U0.x + (P.y -U0.y) * (U1.x - U0.x) / (U1.y - U0.y);
                if (x > P.x)
                    inside = not inside;
            }
        }
        return inside;
    }
```

上述代码的一种稍微不同的变体是计算x-P.x，将各项组合成一个分数，并与零比较：

```
dx = ((P.y - U0.y) * (U1.x - U0.x) - (P.x - U0.x) * (U1.y - U0.y)) /(U1.y - U0.y);
if (dx > 0)
    inside = not inside;
```

其中的分子可以扩展为P.y * (U1.x - U0.x) -P.x * (U1.y - U0.y) +(U0.x * U1.y - U1.x * U0.y)，但是这要求 4 次乘法和 5 次加法。而前面的算法中的分子仅要求 2 次乘法和 5 次加法。更重要的是浮点除法更加费时。利用如下的方法可以避免除法：

```
dy = U1.y - U0.y;
numer = ((P.y - U0.y) * (U1.x - U0.x) - (P.x - U0.x) * dy) * dy;
if (numer > 0)
    inside = not inside;
```

或者，用与零的比较来代替额外的乘法：

```
dy = U1.y - U0.y;
numer = (P.y - U0.y) * (U1.x - U0.x) - (P.x - U0.x) * dy;
if (dy > 0) {
    if (numer > 0) inside = not inside;
} else {
    if (numer < 0) inside = not inside;
}
```

最后的一种变体利用了知道哪一个顶点位于射线之上的知识，并据此进行优化：

```
bool PointInPolygon(Point P, Polygon G)
{
    bool inside = false;
    for (i = 0, j = G.N-1; i < G.N; j = i, i++) {
        U0 = G.V(i);  U1 = G.V(j);

        if (P.y < U1.y) {
            // U1 above ray
            if (U0.y <= P.y) {
                // U0 on or below ray
                if ((P.y - U0.y) * (U1.x - U0.x) > (P.x - U0.x) * (U1.y - U0.y))
                    inside = not inside;
            }
        } else if (P.y < U0.y) {
            // U1 on or below ray, U0 above ray
            if ((P.y - U0.y) * (U1.x - U0.x) < (P.x - U0.x) * (U1.y - U0.y))
                inside = not inside;
        }
```

```
    }
    return inside;
}
```

上述伪码适当地将点分类为严格位于多边形之内或者严格位于多边形之外的点。然而，位于边界上的点有时被归类为位于多边形之内的点，有时被归类为位于多边形之外的点。图 13.15 显示了一个三角形和两个位于边界上的点 P 和 Q。点 P 被归类为位于多边形之内的点，点 Q 被归类为位于多边形之外的点。当两个多边形共用一条边时，应用程序可能需要这种方式。一个点只能位于一个多边形内或者另一个多边形内，而不能同时位于两个多边形内。在要求任何边界上的点都归类为多边形内的点的应用程序中，可以修改上述的伪码，以便当边与射线的交点刚好位于测试点上时，可以处理这种情形。

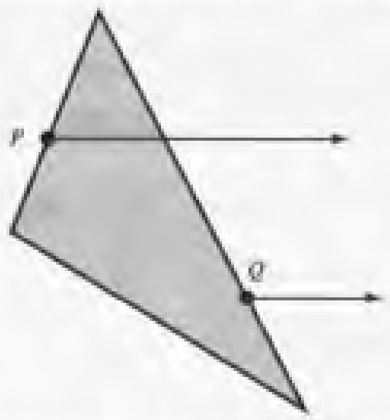


图 13.15　类似 P 的在多边形"左边"的边上的点归类为在多边形内；类似 Q 的在多边形"右边"的边上的点归类为在多边形外

下面让我们更清楚地看看分类问题。右面的边上的点归类为多边形外面的点，其原因与改变表示内部/外部状态的布尔变量值的选择有关。Maynard 和 Tavernini（1984）提出了处理这一问题的不同的方法，这种方法利用直线分解的思想。这种方法之所以令人感兴趣，是因为它能有效地提供能在各维中递归的构造。也就是说，一个 n 维的问题可简化为求解 $(n-1)$ 维的问题。一条边与指定的直线之间的每一个交点都用标记 $\{o, i, m, p\}$ 来表示。标记 i 叫 inside 标记，如果横切相交的交点是边的内点，则使用该标记。标记 m 叫做 minus 标记，如果横切相交的交点是边的一个端点，而另一个端点位于直线的负侧，则使用该标记。标记 p 叫做 plus 标记，如果横切相交的交点是边的一个端点，而另一个端点位于直线的正侧，则使用该标记。标记 o 叫做 outside 标记，如果边与直线重合，则使用该标记来表示两个端点。

检测多边形的每一条边与直线的相交情况。交点第一次出现时，必须修改当前的标记。特别地，当两条或多条边共用一个位于直线上的顶点时，更是如此。假设任何点的初始标记都是 o。如果该点再次出现，并得到一个更新的标记，则将这个标记与老标记组合起来，形成该点的新标记。表 13.1 提供了更新的信息。行对应于老标记，列对应于当前边的更新标记，相应的行与列所确定的项就是新的标记。具有群理论知识背景的读者将注意到 $\{o, i, m, p\}$ 就是 Klein－4 群，而这个表标识该群的运算符。标记 o 是标识元素，并且每一个元素都是其自身的逆。

表 13.1 边与直线相交的标记是 o, i, m 和 p。本表用于更新位于相交点的当前标记。老的标记按行列出。与当前边相交的修正标记按列列出。与点相交的新的标记就是行与列对应的项

老的标记	更新的标记			
	o	i	m	p
o	o	i	m	p
i	i	o	m	p
m	m	p	o	i
p	p	m	i	o

作为一个例子，考虑图 13.14 中显示的多边形。分析位于 P_1 处的射线。这条射线的法线选为 $(0,1)$。边 $\langle 1,2\rangle$ 与射线相交于顶点 2，并且边位于射线的负侧，因此交点的更新标记为 m。默认的初始标记为 o。表 13.1 中的 (o,m) 项为 m，因此顶点 2 的标记设为 m。边 $\langle 2,3\rangle$ 与射线也相交于顶点 2。由于这条边也位于射线的负侧，因此更新的标记也是 m。表中 (m,m) 项为 o。类似的分析将产生图 13.16 中显示的最后标记。注意，顶点 10 处的标记为标记 i，是因为两条边共享该顶点，它们分别具有标记 m 和 p，表中 (m,p) 项为 i。

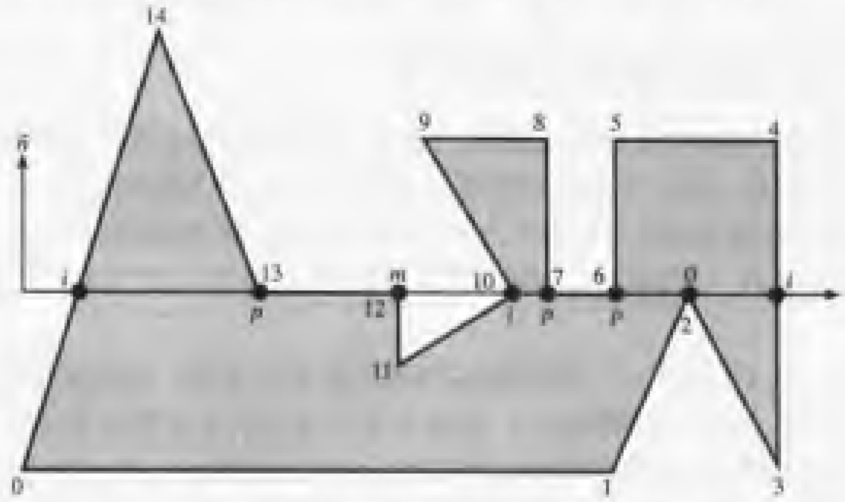


图 13.16 图 13.14 中包含 P_1 的水平线的点的标记

点标记可用于标记直线分解的区间，每一个区间都具有取自 $\{o,i,m,p\}$ 的标记。具有标记 o 的区间位于多边形之外。具有标记 i 的区间位于多边形之内。具有标记 m 的区间与多边形的边重合，并且在边上的多边形的内点位于直线的负侧。具有标记 p 的区间与多边形的边重合，并且在边上的多边形的内点位于直线的正侧。Maynard 和 Tavernini（1984）提出的简单而聪明的方法是利用表 13.1 来从顶点标记产生区间标记。包含 $+\infty$ 的半无穷区间显然位于多边形之外，并开始于一个标记 o。该区间的左端点位于多边形的对右面的边上，该点的标记为 i。最后一个区间标记是 o，并用来选择表 13.1 的行。该点的标记为 i，并用来选择表 13.1 的列。表项 (o,i) 是 i，并成为下一个区间的标记。注意，实际上，这个区间位于多边形内。下一个位于顶点 2 的点的标记是 o。表项 (i,o) 为 i，在左边与顶点 2 直接相邻的区间的标记为 i（仍然位于多边形内）。图 13.17 显示了另一个区间的标记。P_1 的分类就变成了确定直线分区的哪一个区间包含这个点，并合理地解释其标记的问题。如果 P_1 是一个标记为 o 的开放区间，那么这个点位于多边形之外。否则，P_1 位于多边形之内或

者位于多边形的边界上。虽然可以针对这条直线的所有区间建立标记，但我们仅需要射线的标记，因此并不需要计算P_1左边的点的标记。

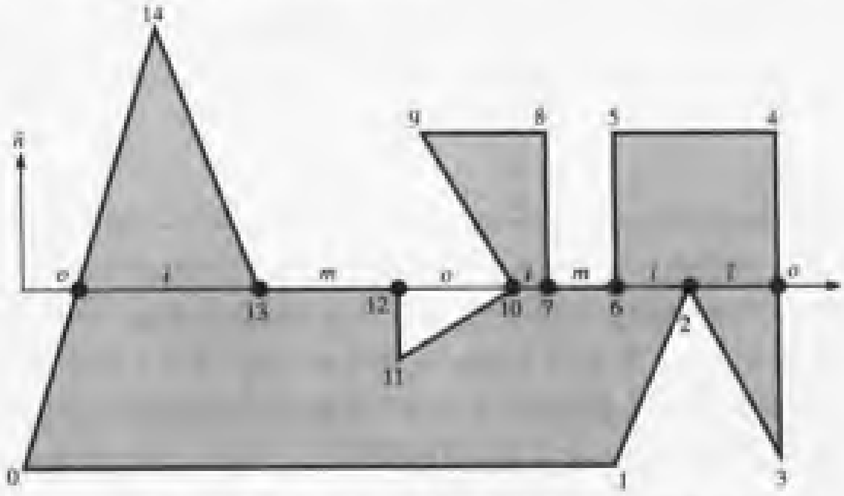


图 13.17　图 13.14 中包含P_1的水平线的区间的标记

13.3.4　点在多边形内的快速检测法

点在多边形内的检测可能是次线性的，然而要求对多边形进行一定的预处理。一种有用的检测是基于将在 13.9 节中介绍的将多边形水平分解为梯形的方法。这种分解是$O(n \log n)$级别的，但是为支持允许进行点在多边形内的检测的分解而建立的直接数据结构的时间级别是$O(\log n)$。下面给出了这种数据结构的简要说明。在 13.9 节中可以找到更详细的内容。

顶点的y值，y_i（$0 \le i < n$），用来将多边形分解为水平条。排序后的y值，表示为y_{i_j}（$(i_0, \ldots, i_{n-1})$（$(0, \ldots, n-1)$的排列），对应于分开这些水平条的水平直线。每一个条本身包含一系列的有序梯形。多边形就是所有条的全部梯形的联合。对y值排序需要$O(n \log n)$级别的时间，尽管任何特定的多边形可能仅有极少量不同的y值。通过对每一条内的梯形的左边和右边线段进行排序，就能对梯形进行排序。这些边界不会重叠，因此可以很好地定义次序。排序的时间也是$O(n \log n)$级别的。使用二叉树进行搜索的数据结构允许被动态构建。一棵树用于排序水平条，另一棵树用于排序条内的梯形。点在多边形内的检测包括搜索第一棵二叉树以定位包含输入点的y值的条，然后搜索第二棵二叉树以定位包含这个点的梯形（如果该梯形存在）。如果没有找到梯形，则这个点必定位于多边形之外。

13.3.5　栅格方法

Haines 和 Heckbert（1994）提出了这种方法。为多边形建立一个轴对齐有界箱。给这个箱子加上栅格，并将多边形栅格化。根据栅格格子与多边形的关系，将它们标记为完全在多边形内、完全在多边形外或者不确定。不确定的格子被指派为与格子相交的边列表，而且格子的一个角被相应地标记为在多边形内或在多边形外。

当点完全位于多边形内或者完全位于多边形外时，栅格数据结构支持点在多边形内的检测的常数时间。如果点位于一个不确定的格子内，那么检测连接检测点和被标记角的线

段与格子的边列表中所有边的相交。根据交点个数的奇偶性，以及被标记的角位于多边形内或外，就能确定检测点位于多边形内还是多边形外。这当然是点在多边形内的检测，但是仅适用于不确定的格子。由于仅与局部相关，因此计算交点的次数大大少于计算与所有多边形的边的交点的算法。这种包含性检测是$O(1)$级的，但是需要为每一个$n \times m$的格子付出$O(nm)$级的内存，并为栅格化多边形和分类栅格格子付出$O(nm)$级的时间。

Haines 注意到，当多边形的边横过（或者几乎横过）一个格子的角时，必须小心处理。角是不能分类的。他给出了数值选项：处理数值精度和拓扑问题，重新栅格化一个稍做修改（比如说，抖动）的有界箱，希望可以去除该问题，或者标记格子的所有边并利用连接测试点和格子边的水平或者垂直线段。如果选择处理数值精度和拓扑问题，可能只需利用点在一般多边形内的检测并处理相同的问题，可以节省内存并减少格子方法的预处理所需的时间。重新栅格化有界箱也是很费时的，特别是需要经常重新栅格化（或者更差的情形，对每一个检测点都要重新栅格化）时，就更是如此。最后的一个建议最有吸引力，当点落入一个不确定的格子内时，算法所需的时间依然很少。

不论如何处理这个问题，在射线跟踪这类点在多边形内的检测是瓶颈的应用程序中，栅格方法都很有用，对于这类问题，你可能愿意用大量的内存和预处理时间来获取$O(1)$级别的检测时间。但是，如果应用程序更趋向于实时应用，并且内容很拮据，那么$O(n)$时间级别的检测算法可能是最好的选择。

13.4 点在多面体内的检测

图形学应用中的一个常见问题是确定一个点是否位于一个多面体内。解决这一问题的思想与 13.3 节中的思想相似。如果用空间分区二叉树来表示多面体，13.2.3 节讨论了如何确定一个点是否位于多面体之内或者多面体之外。我们在此讨论没有对多面体进行任何预处理的方法。

13.4.1 点在四面体内的检测

考虑一个点P和一个具有不共面顶点V_i（$0 \leqslant i \leqslant 3$）的四面体。为了便于讨论，我们假设顶点是有序的，从而使以这种次序的列为$V_i - V_0$（$0 \leqslant i \leqslant 2$）的 3×3 的矩阵 **M** 具有正的行列式。具有这种次序的规范四面体的顶点分别为$V_0 = (0, 0, 0)$，$V_1 = (1, 0, 0)$，$V_2 = (0, 1, 0)$和$V_3 = (0, 0, 1)$。三角面的指向外的法线向量为$\vec{n}_0 = (V_1 - V_3) \times (V_2 - V_3)$，$\vec{n}_1 = (V_0 - V_2) \times (V_3 - V_2)$，$\vec{n}_2 = (V_3 - V_1) \times (V_0 - V_1)$和$\vec{n}_3 = (V_2 - V_0) \times (V_1 - V_0)$。法线为$\vec{n}_i$的面是顶点$V_i$所对的面，且包含顶点$V_{3-i}$。

如果点P位于每一个面平面$\vec{n}_i \cdot (X - V_{3-i}) = 0$的负侧，那么它位于四面体之内。也就是说，当$\vec{n}_i \cdot (P - V_{3-i}) < 0$对所有$i$都成立时，点$P$位于四面体之内。如果$\vec{n}_i \cdot (P - V_i) > 0$对至少一个$i$成立，那么点$P$位于四面体之外。$P$也可能位于四面体的边界上，在这种情形中，$\vec{n}_i \cdot (P - V_i) \leqslant 0$ 对所有i都成立且至少有一个i使等式成立。如果等式出现一次，那么点P位于一个面上。如果等式出现两次，那么点P位于一条边上。如果等式出现三次，那么点P位于一个顶点上。等式不可能出现 4 次。

点也可以用重心坐标表示为$P=\sum_{i=0}^{3}c_iV_i$，其中$\sum_{i=0}^{3}c_i=1$。如果$0<c_i<1$对所有i都成立，那么点P位于四面体之内。系数c_3可用如下的方法计算：

$$P-V_0=(c_0-1)V_0+\sum_{i=1}^{3}c_iV_i=\sum_{i=1}^{3}c_i(V_i-V_0)$$

从而$\vec{n}_3\cdot(P-V_0)=c_3\vec{n}_3\cdot(V_3-V_0)$。类似的构建可应用于$c_0$, c_1和c_2，可得到

$$c_0=\frac{\vec{n}_0\cdot(P-V_3)}{\vec{n}_0\cdot(V_0-V_3)}\quad c_1=\frac{\vec{n}_1\cdot(P-V_2)}{\vec{n}_1\cdot(V_1-V_2)}$$

$$c_2=\frac{\vec{n}_2\cdot(P-V_1)}{\vec{n}_2\cdot(V_2-V_1)}\quad c_3=\frac{\vec{n}_3\cdot(P-V_0)}{\vec{n}_3\cdot(V_3-V_0)}$$

如果点P位于四面体之内，那么其额外表示并不重要。上述分数的分母都是一样的负值。如果所有的$c_i>0$，则点位于多面体之内，在这种情形中，我们要求所有的分子为负数。

13.4.2 点在凸多面体内的检测

点在四面体内的检测问题中所用到的点在哪一侧的检测可以自然地扩展，用于确定一个点是否位于一个凸多面体内。设面包含于平面$\vec{n}_i\cdot(X-V_i)=0$内，其中V_i是这个面的一个顶点，$\vec{n}_i$是这个面的指向外面的法线向量。如果$\vec{n}_i\cdot(P-V_i)<0$对所有i都成立，则点P位于多面体内。如果$\vec{n}_i\cdot(P-V_i)>0$对某些i成立，那么该点位于四面体之外。如果$\vec{n}_i\cdot(P-V_i)\leqslant 0$对所有$i$都成立且至少有一个$i$使等式成立，那么该点位于四面体的边界上。如果等式出现一次，那么点P位于一个面上。如果等式出现两次，那么点P位于一条边上。如果等式出现三次或者三次以上，那么点P位于一个顶点上。在最后一种情形中，等式出现的次数就是共用该顶点的面的个数。

这种算法的级别为$O(n)$，其中n为顶点的个数，因为面的数目也是$O(n)$。简单的实现为

```
bool PointInPolyhedron(Point P, ConvexPolyhedron C)
{
    for (i = 0; i < C.numberOfFaces; i++) {
        if (Dot(C.face(i).normal, P - C.face(i).vertex) > 0)
            return false;
    }
    return true;
}
```

一种渐近较快的方法

由于增加了三维空间的复杂性，因此处理支持$O(\log n)$级问题的凸多边形的索引二分法并不适用于凸多面体。然而，一种类似于二分凸多边形的上半部分和下半部分的方法确实可以扩展到三维空间，只是需要进行一些消耗$O(n)$级时间的预处理。即便如此，当需要进行许多的点在凸多面体内的检测时，这种方法还是非常有用的。

其中的预处理可分为两步。第一步是迭代处理各个顶点，并计算多面体的轴对齐有界箱。这个箱子用来进行快速排除。也就是说，如果P位于有界箱之外，那么不可能位于多面体之内。第二步是迭代处理多面体的各个面。产生凸多面体的两个xy平面网格。将指向外面的法线的z分量为正的面投影到一个平面网格上，该平面网格称为上平面网格。将指

向外面的法线的z分量为负的面投影到另一个平面网格上，该平面网格称为下平面网格。指向外面的法线的z分量为零的面与此无关，可以忽略。

这两个平面网格由凸多边形组成（凸多面体的面都是凸的）。给定一个点P，检测该点的(x, y)分量是否位于上平面网格内。如果它位于网格外，则点P不可能位于多面体内。如果它位于网格内，则必须计算包含这个点的凸多边形。点P的(x, y)分量必定包含于下平面网格内。还要计算这个网格内包含点P的凸多边形。包含点P的常数(x, y)的直线与对应于两个平面凸多边形的多面体的面相交。现在，我们仅需确定点P是否位于包含于多面体内的线段上——这是很简单的问题。

其中的一个技术问题是，如何确定位于平面网格上的哪一个凸多边形包含指定的点。网格上的每一条边都由一个或者两个多边形共享。仅由一个多边形共享的边构成了网格边界——该边界本身就是另一个凸多边形，因为凸多面体在平面上的平行投影是一个凸多边形。定位一个点是否位于一个由网格定义的平面的某一部分的问题叫做平面点的定位问题。

定位包含凸多面体的一种简单算法是在网格上进行线性步进。选取一个初始的多边形，检测点P在该多边形内的包含性。如果该点位于多边形内，那么这个多边形就是包含点的多边形，算法结束。如果该点不位于多边形内，那么选取当前多边形的一条与从多边形的中心（多边形的顶点的平均值）到P的射线相交的边。这条射线给出了寻找P行进方向的一般方法。如果射线通过一个顶点，则选取任一个共用该顶点的边。一旦选取了一条边，下一个要访问的多边形就是共用这条边的另一个多边形。伪码为

```
int LinearWalk(Point P, int N, ConvexPolygon C[N])
{
    index = 0;
    for (i = 0; i < N; i++) {
        // at most N polygons to test
        if (P is contained in C[index])
            return index;
        Point K = C[index].center;  // ray origin
        Point D = P - K;  // ray direction
        for (each edge E of C[index]) {
            if (Ray(C, D) intersects E) {
                index = IndexOfAdjacent(C[index], E);
                break;
            }
        }
    }
    return -1;  // P not in mesh, return an invalid index
}
```

对于位于网格上的n个多边形，该算法是$O(\sqrt{n})$级的。基于大小为$m \times m$的三角形网格的观察来确定次序。该网格具有$n = m^2$个三角形。一条线性路径，比如一行、一列或者一条对角线包含$O(m) = O(\sqrt{n})$个三角形。

有一种渐近快速的方法（Kirkpatrick 1983）可用于处理平面点的定位问题。这种方法要求用基于图形的独立集来建立嵌套的凸多边形的层次。其基本结论是n个顶点的平面网格可在$O(n)$时间和空间级内预处理完成，因而点的定位问题需要$O(\log n)$级的时间。这里不进行这种算法的深入探讨，O'Rourke（1998）讨论了这种算法的一些细节。

13.4.3 点在一般多面体内的检测

针对一般多边形的点在多边形内的检测算法，可以非常明确地扩展到三维的情形。多面体必须分区为一个有界的内部区域和一个无界的外部区域。原点位于检测点P且方向为$\vec{d}=(1,0,0)$的射线与多面体的面相交。要计算交点的数量。假设该射线仅与多面体相交于面的内点，交点数量的奇偶性说明了位于里面还是外面。如果奇偶性为奇，则点位于里面。否则，奇偶性为偶，点位于外面。

然而，当射线与多面体相交于顶点或者边的内点时，会引起与二维情形一样的问题。在这种情形中，奇偶性可能会计算错误。处理这种问题的一种方法是利用表示多面体的顶点－边－面表。这种算法进行对多面体的各个面的迭代处理。将每一个处理过的面标记为已访问。如果射线与面相交于边的内点，则共享这条边的相邻的面立即被标记为已访问，这样在迭代中再访问它时，就不再进行相交测试。而且，必须基于射线、通用边和共用这条边的面的局部构形来校验奇偶性。图 13.18 显示了两个相关的构形。确定这种局部构形是一种简单的任务，只需选择各取自一个面且都不位于通用边上的两个顶点，并计算它们位于射线－边平面的哪一侧。三维空间中的射线—相交—边的情形与二维空间中的射线—相交—顶点的情形类似。

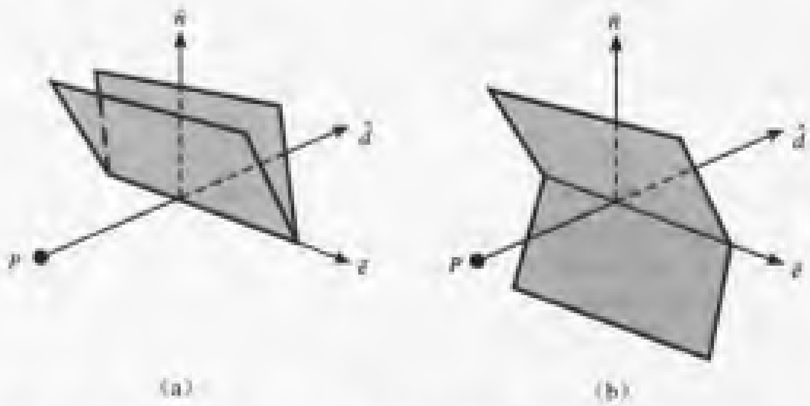


图 13.18 测试射线$P+t\vec{d}$与共用边$\vec{e}$相交于一个内部边上的点时的构形
（a）所有面在边和射线形成的平面的同一边。奇偶性没有变化
（b）所有面在边和射线形成的平面的不同边。奇偶性发生反转

如果射线与顶点V相交，情形就会更加复杂，而且在二维空间中没有对应的情形。其中的问题是决定射线是否在V处穿透多面体，或者是否仅在顶点上相擦而过，使得射线局部地保留在同一区域。特别地，设$V=P+t_0\vec{d}$对某些参数t_0成立。对于一个适当小的数$\epsilon>0$，我们需要确定两条分别对应于参数区间$(t_0-\epsilon,t_0)$和$(t_0,t_0+\epsilon)$的开放线段（不包含端点的线段）是否都位于里面或者外面，在这种情形中，当前的奇偶性都不会改变；或者一个位于里面而另一个位于外面，在这种情形中，当前的奇偶性将被改变。我们可以想像一个非常“粗糙”的顶点，其相邻的面构成一个三角形带，这个三角形带在该顶点的局部空间上任意地扰动，这可能使问题难以处理。然而，事实上，多面体是一个多面网格（参见 9.3.3 节）。如果一个单位球的中心位于顶点V，而且共用V的边重新缩放为单位长度，

那么对应的球面上的点构成一条位于球面上的简单封闭曲线，更精确地说，是构成一条分段定义的曲线，其每一段都是一条大圆弧。以该曲线为界的内部区域对应于位于 V 的内部多面体。射线方向本身可被规整化，并对应于球面上的一个点。当且仅当对应的球面上的点位于由共用 V 的边所定义的球面多边形之内时，射线在点 V 穿过多面体。如图 13.19 所示，这并不是前面讨论过的点在多边形的检测，但是可以构建一个与处理平面多边形的算法类似的算法，用来处理球面多边形。

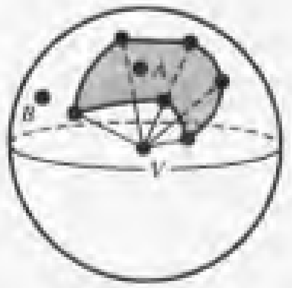


图 13.19 球面多边形由共用测试射线相交的顶点 V 的边确定。如果点 A 与射线的方向相符，则射线穿过多面体。如果点 B 与射线的方向相符，则射线不穿过多面体

另一种处理精细比例的方法是利用随机化方法。其思想是产生随机方向，直到找到一条仅与面相交于内点的射线。其伪码为

```
bool PointInPolyhedron(Point P, ConvexPolyhedron C)
{
    parity = false;
    i = 0;

    while (i < C.numberOfFaces) {
        Point D = GenerateRandomUnitVector();
        for (i = 0; i < C.numberOfFaces; i++)  {
            if (Ray(P, D) intersects C.face(i))  {
                if (intersection point is interior to face))
                    parity = not parity;
                else  // bad ray, try a different one
                    break;
            }
        }
    }

    return parity;
}
```

虽然可以期望最外层的循环最终会终止，但是并不清楚出现多少次迭代才会终止。有一种变体是用 for 循环代替 while 循环，并在 for 循环中指定一个可能出现的最大的循环次数。如果在所有的这些迭代中都没有发现合适的射线，那么可以调用一个慢得多的算法，比如，Paeth（1995）“用球面多边形来进行点在多面体内的检测”中的算法。这种方法要求计算实体角度，而且在运算中使用了反三角函数调用，因此性能很差。

13.5　与多边形有关的布尔运算

在计算机图形学中经常遇到的一个问题是如何计算两个多边形 A 和 B 之间的交集，这其实是一组通常称为多边形的布尔运算中的一个问题。假设每一个多边形自身都不相交，即没有一条边与另一条边横切相交，但是边允许相会于顶点。一般地，假设每一个多边形都封闭一个互连的有界区域，区域内可能存在洞。我们并不要求多边形具有边界，边允许为线段、射线或者直线。我们也并不要求互连性。例如，两个分离的三角形可被看做一个多边形的一部分。多边形的一般定义使我们能用非常简单的方法来实现除相交之外的其他运算。

平面必定被多边形分区为两个分离的区域，即一个内部区域和一个外部区域。多边形（可能没有边界）的每一条线形对象都具有与其相关的法线向量。法线所指向的区域被标记为“外部”；相反的区域被标记为“内部”。相应地，如果线形对象表示为有向的，那么当沿着线形对象按指定的方向前进时，内部区域位于左边，而外部区域位于右边。图 13.20 中显示了 3 个多边形，一个是有边界的凸形，一个是有边界的非凸形，而另一个没有边界。

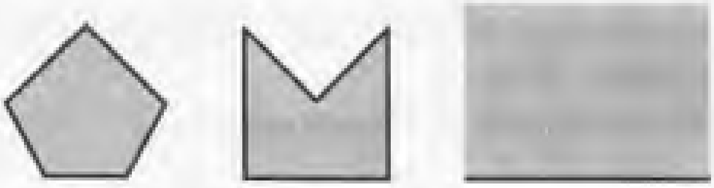


图 13.20　有界和无界多边形将平面分成内部和外部两个区域，内部区域显示为灰色。右边的无界多边形是一个半空间，以一条单独的直线作为区域的边界

13.5.1　抽象运算

多边形的布尔运算包括如下的运算。

1. 反

这种操作反转内部和外部区域的标记，内部区域变成外部区域，外部区域变成内部区域。如果在存储多边形时，明确地保存了边的法线，那么反运算可用法线的符号变化来实现。如果在存储多边形时，保存了边的方向，那么反运算可通过反转所有边的方向来实现。图 13.21 显示了一个多边形和它的反。

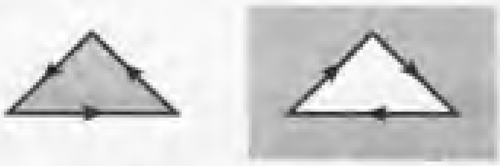


图 13.21　一个多边形与其反形，内部区域显示为灰色，边显示出适当的方向使得内部总是在边的左边

其余的几种运算将用显示在图 13.22 中的两个多边形来说明，即一个反 L 形多边形和一个五边形，其中显示了边的方向，表示两个多边形的内部区域都是有界的。

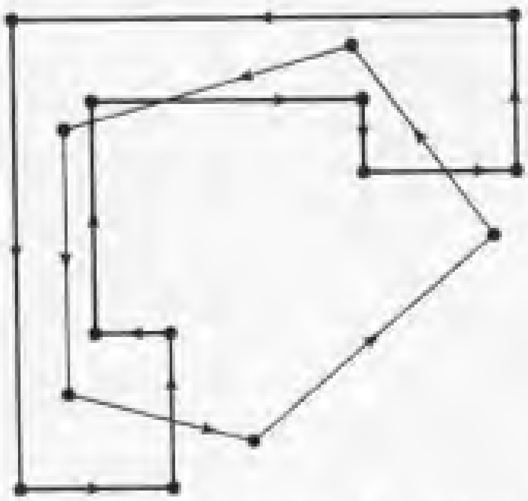


图 13.22　两个内部区域是有界区域的多边形

2. 交

两个多边形的交就是一个多边形的内部区域与另一个多边形的内部区域的相交。图 13.23 显示了两个多边形的交。多边形的顶点和边显示为黑色的，而交是灰色的。根据本节开始的定义，交也是一个多边形，它由两个对象组成，每一个都是一个简单的多边形。

3. 并

两个多边形的并是一个多边形，其内部区域是原来的两个多边形的内部区域的联合。图 13.24 显示了两个多边形的并。多边形的顶点和边显示为黑色的，而并是灰色的。

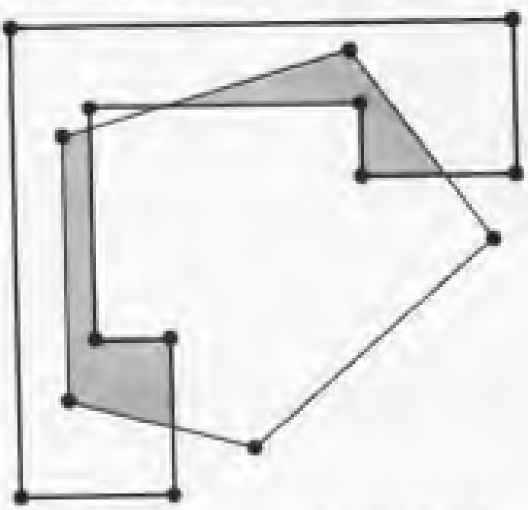


图 13.23　两个多边形的交显示为灰色

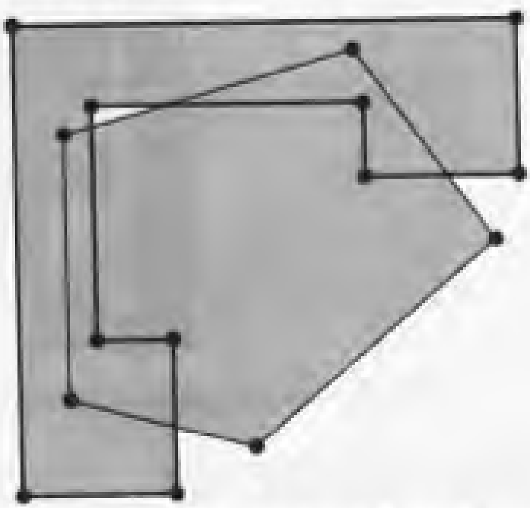


图 13.24　两个多边形的并显示为灰色

4. 差

两个多边形的差是一个多边形，其内部区域是原来的两个多边形的内部区域的差。多边形的次序很重要。如果 A 是第一个多边形的内部区域的点集，而 B 是第二个多边形的内部区域的点集，那么差 $A \setminus B$ 就是属于 A 且不属于 B 的点集。图 13.25 显示了两个多边形的差，即反 L 形多边形减去五边形。多边形的顶点和边显示为黑色的，而差是灰色的。

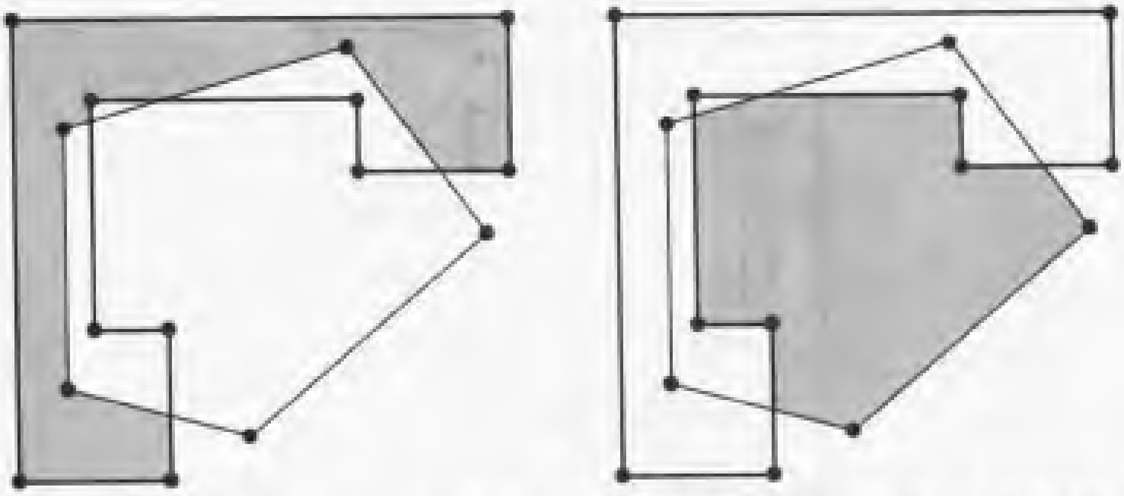


图 13.25　两个多边形的差：(a) 反 L 形多边形减去五边形，(b) 五边形减去反 L 形多边形

5．异或

两个多边形的异或是一个多边形，其内部区域是原来的两个多边形的差的并。如果 A 是第一个多边形的内部区域的点集，而 B 是第二个多边形的内部区域的点集，那么异或的内部区域就是点集 $(A \setminus B) \cup (B \setminus A)$。图 13.26 显示了两个多边形的异或。多边形的顶点和边显示为黑色的，而异或是灰色的。

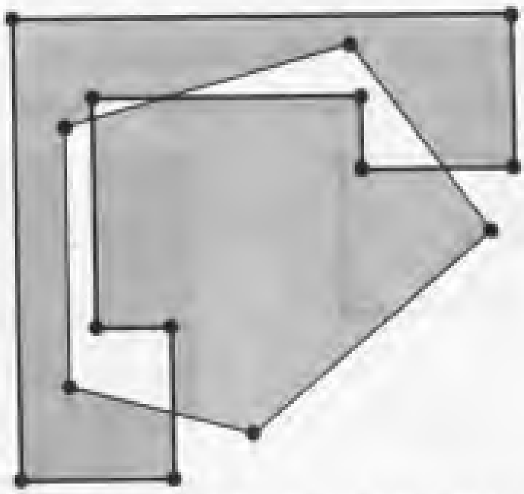


图 13.26　两个多边形的异或显示为灰色。该多边形是图 13.25 中显示的两个差的并

13.5.2　两种基础运算

虽然布尔运算可以根据每一种运算的定义来实现，但是仅需实现反和交运算。其他的布尔运算可用这两种基本运算来定义。

1．反

多边形 P 的反表示为 $\neg P$。这个一元运算符的优先级高于下面的其他运算符。

2．交

多边形 P 和多边形 Q 的交表示为 $P \cap Q$。

3. 并

多边形P和多边形Q的并表示为$P \cup Q$，并可利用集合的德摩根律来计算

$$P \cup Q = \neg(\neg P \cap \neg Q)$$

4. 差

多边形P和多边形Q的差，也就是从Q减去P，表示为$P \setminus Q$，并可用下式来计算

$$P \setminus Q = P \cap \neg Q$$

5. 异或

多边形P和多边形Q的异或表示为$P \oplus Q = (P \setminus Q) \cup (Q \setminus P)$，并可用下式来计算

$$P \oplus Q = \neg((\neg(P \cap \neg Q)) \cap (\neg(Q \cap \neg P)))$$

13.5.3 使用空间分区二叉树的布尔运算

已经使用了多种方法来进行多边形的布尔运算。一种流行的方法是直接利用空间分区二叉树来实现布尔运算。这种思想自然地扩展到三维空间，在三维空间中可对多面体进行布尔运算（参见 13.6 节）。下面讨论两种基础运算。

1. 反

可以简单地实现多边形的反运算。假设多边形的数据结构保存了边，与讨论多边形的空间分区二叉树表示的情形一样，通过反转每一条边的次序来实现反运算。

```
Polygon Negation(Polygon P)
{
    Polygon negateP;
    negateP.vertices = P.vertices;
    for (each edge E of P) do
        negateP.Insert(Edge(E.V(1), E.V(0)));
    return negateP;
}
```

表示多边形的空间分区二叉树可用如下的伪码来取反：

```
BspTree Negation(BspTree T)
{
    BspTree negateT = new BspTree;

    for (each edge E of T.coincident)
        negateT.coincident.Insert(Edge(E.V(1), E.V(0)));

    if (T.posChild)
        negateT.negChild = Negation(T.posChild);
    else
        negateT.negChild = null;

    if (T.negChild)
        negateT.posChild = Negation(T.negChild);
    else
```

```
        negateT.negChild = null;

    return negateT;
}
```

2. 交

可用非常简单的方式来实现多边形的交。如果A和B是多边形，A的每一条边与B都相交。这些边位于B内的任何部分都保留为相交所得的多边形的一部分。类似地，B的每一条边与A都相交。这些边位于A内的任何部分都保留为相交所得的多边形的部分。图 13.27 说明了这一点。虽然这个算法很简单，但是，其效率并没有想像的那么好。如果多边形A有n条边，多边形B有m条边，那么边一边相交检测的次数为nm，因此，算法的时间复杂度为$O(nm)$(时间的二次函数)。由于位于分区直线的一侧的A的边不需要与位于另一侧的B的边进行比较，因此利用空间分区二叉树可以减少比较的次数。

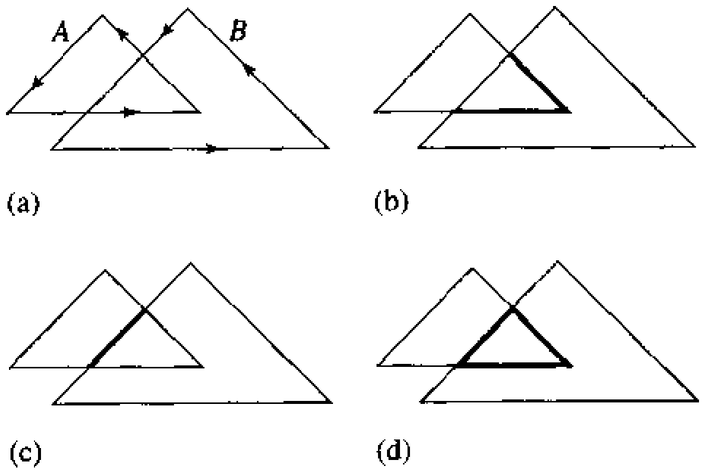


图 13.27　两个三角形的交：(a) 两个三角形A和B (b) A的相交部分的边在B内 (c) B的相交部分的边在A内 (d) $A\cap B$是所有相交的边的总和

正如图 13.27 所说明的，每一个多边形的边必须与另一个多边形的内部区域相交。位于内部的任何子边都成为两个多边形的交的一部分。

伪码为

```
Polygon Intersection(Polygon P, Polygon Q)
{
    Polygon intersectPQ;

    for (each edge E of P) {
        GetPartition(E, Q, inside, outside, coincidentSame, coincidentDiff);
        for (each S in (inside or coincidentSame))
            intersectPQ.Add(S);
    }

    for (each edge E of Q) {
        GetPartition(E, P, inside, outside, coincidentSame, coincidentDiff);
        for (each S in (inside or coincidentSame))
            intersectPQ.Add(S);
```

```
        }

        return intersectPQ;
    }
```

这种方法的核心是对一个多边形的边 E 进行分区，即用另一个多边形来对这条边分区。函数GetPartition建立了与指定的多边形相交的边 E 的 4 个线段集合，其中至少有一个集合是非空的。两个集合对应于位于多边形之内的线段或者多边形之外的线段。另外两个集合对应于与这个多边形的边重合的线段，其中一个集合（coincidentSame）存放与边同向的线段，另一个集合（coincidentDiff）保存与边反向的线段。正如在 13.1.4 节中所讨论的，利用空间分区二叉树来表示多边形，通过空间分区二叉树函数GetPartition可以高效地进行分区。

不要在对应多边形的边的相反方向中包含重合的线段，使得寻找相交时仅需计算与正面积的相交。例如，图 13.28 显示了两个相交为线段的多边形。前面显示的计算相交的伪码将报告这两个多边形不相交。当然，如果要在这个例子中使相交算法返回一条线段，就可以包括另一组重合边的集合。图 13.29 显示了另一个稍微复杂一些的例子。列出的伪码返回相交的三角区域，而不是悬在三角形上的实际的边。如果修改代码使其包含所有的重合线段，那么相交算法将返回三角形和实际的边。有的应用程序可能需要相交的真正集合；另外一些应用程序可能只需要对象与正面积的相交，应根据实际需要来实现。

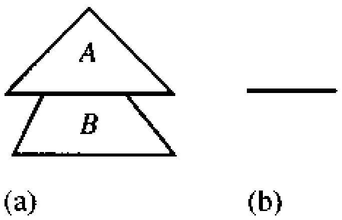


图 13.28　（a）伪码报告不相交的两个多边形（b）实际的交集，一条线段

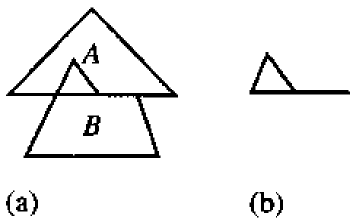


图 13.29　（a）两个多边形（b）它们的真实交集

这里讨论的交运算支持所谓的“锁眼边”，即两条边共线且具有相反的方向。锁眼边一般用于表示具有用单个顶点/边的集合来表示的洞的多边形。图 13.30 显示了一个具有一个洞的多边形，并用锁眼来表示这个洞。然而，应该注意，一个多边形与一个具有锁眼的多边形的相交在理论上可能是一个简单多边形，但是被构建为多个简单多边形的一个并。图 13.31 说明了一个矩形与前一个图所显示的锁眼多边形的交。交集包含两个相邻但方向相反的边，在后续的步骤中可以删除这些边。

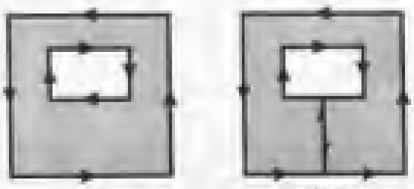


图 13.30　(a) 多边形有一个要求两组顶点/边的洞 (b) 可仅由一组顶点/边确定的锁眼

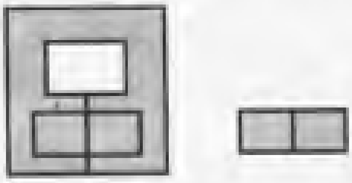


图 13.31　矩形和锁眼多边形的交

3. 其他的布尔运算

利用反和交作为基本运算，其余的几种布尔运算可以实现如下。

```
Polygon Union(Polygon P, Polygon Q)
{
    return Negation(Intersection(Negation(P), Negation(Q)));
}

Polygon Difference(Polygon P, Polygon Q)
{
    return Intersection(P, Negation(Q));
}

Polygon ExclusiveOr(Polygon P, Polygon Q)
{
    return Union(Difference(P, Q), Difference(Q, P));
}
```

正如前面所提到的，从边列表中过多地进行交运算可能导致很差的性能。另一种方法是从原来多边形的边列表和空间分区二叉树开始，并不建立中间的多边形，而只建立最终的多边形。对异或运算的一种简单的改进是利用一种二元运算符 `DisjointUnion` 运算，其运算结果的边是两个输入多边形的边的并。这两个不同的多边形没有共同的边，因此，一个完整的并运算将比这种情形中的运算做更多的事情。

13.5.4　其他算法

基于空间分区二叉树的算法并不是实现多边形布尔运算的惟一选择，特别是对于多边形的交运算而言。所有的方法都具有一个共同点——它们都必须找到一对边的交点。出现交点的边必须被（抽象地）分解成子边。这些方法中所不同的是如何将子边合并成多边形形式的交。

我们简要地介绍一下其他的方法。Weiler-Atherton（WA）算法（Weiler 和 Atherton 1977）

是一种在计算机图形学新闻组中经常会遇到的算法。Foley 等人（1996）中包含了这种算法的详细描述，包括数据结构是如何组织的。然而，我们认为，下面的两种算法是更好的选择，原因如下。Sechrest 和 Greenberg（1981）提出了一种基于扫描线（用一个对象来排序顶点）的算法，我们在此称之为 SG 算法。Vatti（1992）提出了一种稍做修改的 SG 算法，我们称之为 V 算法。后一种算法允许将多边形的交作为一组梯形，而不是一般多边形输出，这对于进行点在多边形内的检测来说是非常有用的。V 算法的关键是将一个多边形水平分解为一些梯形，我们将在 13.9 节中讨论梯形。

计算边一边相交时，性能非常重要。很明显，检查所有可能的边对的相交的算法自然是效率最低的算法。空间分区二叉树算法避免了比较所有的边对，而是利用了分区线所表示的空间排序的性质。SG 算法和 V 算法提供了最高效的检测，因为它们利用了基于扫描线方法的水平排序。

标识处理交运算所得的多边形的子边的是空间分区二叉树算法的一个自然结果。而且，这种算法自然地提供了交运算所得的多边形的顶点分解。然而，一般分解方法会比 SG 算法和 V 算法花费更多的时间来比较更多的边。SG 算法中的子边标识和合并通过双重边来完成。每一条抽象的多边形边都具有两个实例，一个赋予多边形的内部，另一个赋予多边形的外部。这些边的标记将保持存储这类信息。边对的交点被定位，并且包含交点的（所有 4 条）边被分解。这些边基于标记被重新连接，以形成不相交的轮廓线，其中之一对应于交多边形。SG 算法和 V 算法中的子边标识和合并以将多边形分区为水平条为基础，每一条都包含条内的一个梯形表。13.9 节中的材料给出了分区的更多细节。

任何计算多边形的交的算法都关注它将处理什么样的多边形。所有的方法都假设多边形顶点具有某种一致的次序，不论是顺时针方向还是逆时针方向。这种方法是否仅能处理凸多边形？它能否处理具有洞的多边形，或者更一般地，嵌套的多边形？它是否允许多边形具有自身相交的边？它是如何处理重合边的？有些算法声称能“正确”处理重合边，但是从图 13.28 和图 13.29 中可以看出，对重合边的行为的选择是由应用程序指定的。一个应用程序可能恰好仅需要具有正面积的交多边形。在理论上，这里讨论的所有算法都能处理嵌套多边形，但是特定的实现却可能不能处理嵌套多边形。

不同算法的实现可以从网上在线获得。Klamer Schutte 提供了一个基于 WA 算法的实现，但其中对将子边合并为交多边形的处理做了一些修改（Schutte 1995）。该实现要求顶点为顺时针次序，不支持洞，也不支持自身相交的边。

Michael Leonov 提供了一种实现，对 Schutte 的实现做了一些修改，这种实现支持洞（Leonov 1997）。外面的多边形必须是逆时针排序的，而构成洞的内部多边形必须是顺时针方向的。Leonov 还对各种不同的实现进行了非常好的比较，包括分析它们的执行时间以确定收敛次序，以及一个表示实现是否能处理不同类型的多边形的图表。不幸的是，在本书出版时，这一网页不再存在。

Alan Murtha 有一种基于 V 算法的实现，编写得很好，并被许多人下载（Murtha 2000）。Klaas Holwerda 也有一种基于 V 算法的实现（Holwerda 2000）。

两种实现都可以从 Eberly（2001）的源程序网页中得到，其中一种实现利用了多体的概念，另一种实现利用了空间分区二叉树。

13.6　与多面体有关的布尔运算

这个主题被认为是一类通用方法的一部分，这些方法统称为计算实体几何（CSG），用于操作三维空间对象。多面体的布尔运算与多边形的布尔运算的种类相同。我们建议你先阅读 13.5 节，以理解如何对多边形进行布尔运算。假设多面体为分区空间，这样区域的抽象图形可用两种颜色来区分。直观地，每一个不相连的区域都标记为多面体的“内部”或者“外部”。一般地，内部区域是有界的（具有有限的体积），但是我们使用更一般的术语“二色区分”，以允许内部和外部区域都是无界的。例如，在布尔运算中允许出现半空间。基于选取一个法线指向平面的某一侧来标识外部区域。基于选取一个法线指向平面的另一侧来标识内部区域。提供一个二色区分空间和内部区域是有界的多面体在计算实体几何中被称为多面实体（Maynard 和 Tavernini 1984）。

13.6.1　抽象运算

抽象运算包括多面体的反，即反转内部区域和外部区域的标记；两个多面体的交，即两个输入的多面体同时包含的子多面体；两个多面体的并，即由任何一个输入的多面体所包含的多面体；两个多面体的差，即包含在第一个输入的多面体之内，而不包含于第二个输入的多面体之内的多面体；以及两个多面体的异或，即这两个多面体的差的并。与二维的情形一样，反运算和交运算是基础运算。其他的运算可用它们来表示。一个多面体P的反表示为$\neg P$，而多面体P和多面体Q的交表示为$P \cap Q$。多面体的并为$P \cup Q = \neg(\neg P \cap \neg Q)$，多面体的差为$P \setminus Q = P \cap \neg Q$，而多面体的异或为$P \oplus Q = \neg((\neg(P \cap \neg Q)) \cap (\neg(Q \cap \neg P)))$。布尔运算的最小编码方法是：首先实现反和交，然后根据这两种运算来实现其他的运算。

13.6.2　使用空间分区二叉树的布尔运算

利用空间分区二叉树可以非常有效地实现多面体的布尔运算。可以从 Thibault 和 Naylor（1987）；Naylor（1990）；Naylor，Amanatides 和 Thibault（1990），以及 Naylor（1992）中找到这种方法。在本节的讨论中，假设多面体存储在顶点－边－面表中，并且面的顶点是有序的，使得面的法线对应于顶点的逆时针方向。假设法线指向外部区域。

1．反

多面体的反可以通过反转面的顶点的次序来简单地实现，如果存储了法线，则可以将法线乘以–1 使其反向。伪码为

```
Polyhedron Negation(Polyhedron P)
{
    Polyhedron negateP;
    negateP.vertices = P.vertices;
    negateP.edges = P.edges;
    for (each face F of P) {
        Face F';
```

```
        for (i = 0; i < F.numVertices; i++)
            F'.InsertIndex(F.numVertices - i - 1);
        negateP.faces.Insert(F');
    }
    return negateP;
}
```

可以用如下的伪码来反转表示多面体的空间分区二叉树：

```
BspTree Negation(BspTree T)
{
    BspTree negateT = new BspTree;

    for (each face F of T.coincident) {
        Face F';
        for (i = 0; i < F.numVertices; i++)
            F'.InsertIndex(F.numVertices - i - 1);
        negateT.coincident.Insert(F');
    }
    if (T.posChild)
        negateT.negChild = Negation(T.posChild);
    else
        negateT.negChild = null;

    if (T.negChild)
        negateT.posChild = Negation(T.negChild);
    else
        negateT.negChild = null;

    return negateT;
}
```

2. 交

多面体的交也可以用简单的方式来实现。如果 A 和 B 是多面体，A 的每一个面与 B 都相交。位于 B 内的面的任何部分都成为交多面体的一部分。类似地，B 的每一个面与 A 都相交，而且位于 A 内的面的任何部分都成为交多面体的一部分。与二维情形一样，检测面对（每一个面对都有一个面为 A 的 n 个面中的一个，以及一个面为 B 的 m 个面中的一个）的 $O(nm)$ 级算法是非常低效的。利用空间分区二叉树可以减少比较的次数，因为树隐含了空间排序。

伪码列出如下：

```
Polyhedron Intersection(Polyhedron P, Polyhedron Q)
{
    Polyhedron intersectPQ;

    for (each face F of P) {
        GetPartition(Q.bsptree, F, inside, outside, coinside, cooutside);
        for (each face S in (inside or coinside))
            intersectPQ.faces.Insert(S);
    }

    for (each face F of Q) {
```

```
        GetPartition(P.bsptree, F, inside, outside, coinside, cooutside);
        for (each face S in (inside or coinside))
            intersectPQ.faces.Insert(S);
    }
}
```

这种方法的核心是通过将多面体的一个面 F 与另一个多面体的相交来对其分区。实现这一步的函数是 GetPartition，我们已在 13.2.5 节中讨论过这一函数。

3. 其他布尔运算

实现其余的布尔运算的方法与前面介绍过的方法相同。

```
Polyhedron Union(Polyhedron P, Polyhedron Q)
{
    return Negation(Intersection(Negation(P), Negation(Q)));
}

Polyhedron Difference(Polyhedron P, Polyhedron Q)
{
    return Intersection(P, Negation(Q));
}

Polyhedron ExclusiveOr(Polyhedron P, Polyhedron Q)
{
    return Union(Difference(P, Q), Difference(Q, P));
}
```

13.7 凸包

如果对于任何$X, Y \in S$，线段$(1-t)X + tY \in S$对于$t \in [0,1]$都成立，那么集合$S \subset \mathbb{R}^n$为凸集合。这一定义并不要求这个集合是有界的或者封闭的。例如，所有$\mathbb{R}^2$都是凸的。由$x > 0$所定义的$\mathbb{R}^2$内的半平面是凸的。其他的凸集的例子有圆盘和三角形。自然，直线、射线和线段也是凸的。图 13.32 显示了两个集合：图 13.32（a）中的集合是凸的，图 13.32（b）中的集合不是凸的。在三维空间中，凸集合的例子是$\mathbb{R}^3$、半空间、球、椭球、直线、射线、线段、三角形和四面体。

集合 S 的凸包就是包含 S 的最小凸集合。在计算机图形学中特别瞩目的是有限点集的凸包。本节主要讨论点集的凸包的建立。

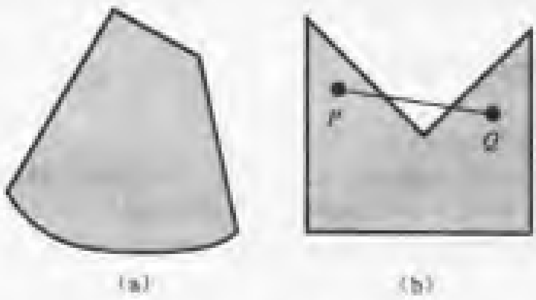


图 13.32　(a) 凸多边形 (b) 非凸多边形，因为连接 P 和 Q 的线段并非完全位于源集内

13.7.1　二维凸包

考虑一个有限点集 S。如果该集合的所有点都共线，那么其凸包就是一条线段。更令人感兴趣的是当至少有三个点不共线时的情形。在这种情形中，凸包是一个被称为凸多边形的多边形所包围的区域。图 13.33 显示了一个点集和其凸包。凸包的顶点必须是原点集的一个子集。将 S 内的凸多边形的顶点标识出来就建立了凸包。

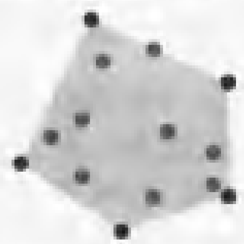


图 13.33　一个点集与其凸包。除了成为凸包的顶点的点显示为黑色外，其余点都显示为深灰色。包显示为浅灰色

已经开发出了计算点集的凸包的数值算法。O'Rourke（1998）中有对这些算法的简介，包括 gift wrapping 算法、quickhull 算法、Graham 算法、增量建立算法，以及分而治之算法等。我们仅讨论上述算法中的最后两种。许多计算几何学书籍都对点集进行限制，以简化凸包的建立和证明。典型的假设包括没有重复的点、没有共线的点、和/或仅有整数坐标的点以允许精确的算术运算。在实际情形中，通常难以满足这些假设，特别是在存在浮点数的系统中。为了提供一个健壮的系统，我们非常注意可能出现的病态问题。

1．增量构建算法

这个算法的思想很简单。给定一个点集 $V_i, 0 \leqslant i < n$，每一个点都插入一个已经建立的前一个点的凸包中。伪码为

```
ConvexPolygon IncrementalHull(int n, Point V[n])
{
    ConvexPolygon hull = {V[0]};
    for (i = 1; i < n; i++)
        Merge(V[i], hull);
    return hull;
}
```

这个问题的核心是如何建立凸多边形 H 和点 V 的凸包，在伪码中，进行这个操作的函数的名字是 `Merge`。如果 V 在 H 之内，合并的一步不做任何事情。但是如果 V 在 H 之外，合并的一步必须找到从 V 出发的刚好与凸包相触的射线。这些射线称为凸包的切线（参见图 13.34）。

切线具有凸包完全位于直线的一侧，至多只有一个顶点或点的边位于这条直线上的性质。在图 13.34 中，上面的切线与当前的包相交于一个点。下面的切线沿着当前的包的边与包相交。切点是从 V 可以看见的所有凸包的顶点的极值点。其余的包顶点被包自己锢囚于 V。在这个图中，下面的切线包含一条端点为 P_L 和 P_L' 的边，但是仅有 P_L 是对 V 可见的。

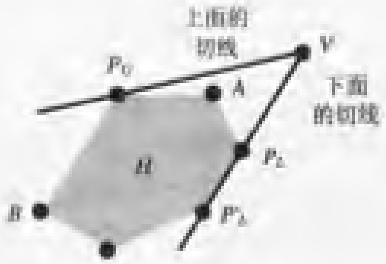


图 13.34　凸包H，H之外的点V，以及包的始于V的切线。上下切点分别标为P_U和P_L

对V可见的条件可以被进一步利用。当前包的边位于由两条对V可见的切线所定义的圆锥之内。这些边位于包含H和V的新包之内。新包包括所有当前包的被钳囚的点，以及由从V到切点的线段所构成的新边。在图 13.34 的例子中，两条边对V是可见的，并且可被排除。新的边为$\langle V, P_U\rangle$和$\langle V, P_L\rangle$。由于边$\langle P'_L, P_L\rangle$完全位于下面的切线上，因此这条边也可被排除并被新边$\langle V, P'_L\rangle$所代替。如果不执行这一步，那么这些点的最终的包将包含共线的边。这样的边可以在后加工步骤中排除。

确定一条边是否可见仅需计算三个点（直接边的两个端点和点V）的次序。在图 13.34 中，直接边$\langle P_L, A\rangle$对V是可见的。三角形$\langle V, P_L, A\rangle$是顺时针次序的。直接边$\langle P_U, B\rangle$对V是不可见的。三角形$\langle V, P_U, B\rangle$是逆时针次序的。直接边$\langle P'_L, P_L\rangle$上只有P_L对V是可见的。三角形$\langle V, P'_L, P_L\rangle$已退化（为一条线段）。这种情形由点积来确定相关的数量。设$\langle Q_0, Q_1\rangle$是一条直接的包边，并定义$\vec{d} = Q_1 - Q_0$和$\vec{n} = -\vec{d}^{\perp}$，这是一条边的指向内部的法线。只要$\vec{n} \cdot (V - Q_0) < 0$，则这条边对$V$是可见的；只要$\vec{n} \cdot (V - Q_0) > 0$，则这条边对$V$是不可见的。如果点积为零，那么只有边的最近的端点对V是可见的。如果$\vec{d} \cdot (V - Q_0) < 0$，那么端点$Q_0$是最近的端点；如果$\vec{d} \cdot (V - Q_0) > \|\vec{d}\|^2$，那么端点$Q_1$是最近的端点。

算法所需的时间级取决于在合并的一步中必须做的工作的数量。在最坏的情形中，`Merge`的每一个输入点都位于当前包的外面，通过迭代包的所有顶点并检测点积条件，可以找到切点。所需的时间级是$O(n^2)$。正因如此，希望以这种方式将初始点排序，使得输入点在大部分时间内都位于当前包内。这是随机算法的思想，输入点被随机地改变序列，以从开始的几个输入点中产生大部分的包。当出现这种情形时，剩下的许多点都非常可能落入当前包的内部。由于并不知道下一个输入点和当前包的关系，必须搜索包的顶点以找到切点。Berg 等人（2000）中讨论了一个随机算法。在 13.11 节中寻找包含点积的最小面积的圆时，也介绍了相同的思想。

一种可以保证为$O(n \log n)$时间级算法的非随机方法实际上对点进行排序，以保证下一个输入点位于当前的凸包之外。对点的排序占用了算法的大部分时间。根据小于操作来对点排序：当$x_0 < x_1$或者$x_0 = x_1$和$y_0 < y_1$时，$(x_0, y_0) < (x_1, y_1)$。这里的讨论假设这些点在开始时已被排序。在实现中，为了不改变原始的点集，这些点极可能被排序并另外存储。在排序之后，重复的点可以被排除。实现排序和比较时，可以使用模糊浮点算术来处理浮点舍入误差可能导致两个（理论上）相等的点变得有一些差别的情形。

这种算法还存有一些关于下一个输入和当前包之间的关系的信息，使得在伪码循环的整个生命期内，切线构造仅是$O(n)$级的。特别地，最后插入的包的顶点就是搜索切点的起始点，并且还可能就是切点。虽然任何一次简单的搜索都可能要求访问当前包上的大量的

点，但同时每一个被访问的点都将位于合并的包内并可被排除。用于每一个被排除的点的平均时间是一个常数，因此在整个循环的生命期内，总的时间为$O(n)$级。

第一个点就是初始的包。当输入点被处理时，一个标记（即type）被维护，以标识包是否是一个点（POINT）、由两个不同的点（LINEAR）表示的一条线段或者一个具有正面积（PLANAR）的凸多边形。在开始时，标记为POINT，因为由包进行排序的初始点就是第一个输入点。通过跟踪记录这个标记，我们可以有效地基于针对较低维（在这种情形中为一维）的凸包算法得到一个针对给定维（在这种情形中为二维）空间的凸包算法。伪码成为

```
ConvexPolygon IncrementalHull(int n, Point V[n])
{
    Sort(n, V);
    RemoveDuplicates(n, V);  // n can decrease, V has contiguous elements
    ConvexPolygon hull;
    type = POINT;
    hull[0] = V[0];
    for (i = 1; i < n; i++) {
        switch (type) {
            case POINT:   type = LINEAR;  hull[1] = V[i];  break;
            case LINEAR:  MergeLinear(V[i], hull, type);   break;
            case PLANAR:  MergePlanar(V[i], hull);         break;
        }
    }
    return hull;
}
```

如果当前的凸包有一个点，那么点的惟一性意味着V[i]不同于包内已有的点。该点被加入当前包中，以构成一条线段，并且type标记做相应的修改。

如果当前的包有两个点（一条线段），函数MergeLinear确定当前的输入点是否与包内的线段位于同一条直线上。如果它们位于同一条直线上，则当前的包被更新并保留一条线段。如果他们不位于同一条直线上，则当前的包和输入点构成一个三角形。在这种情形中，该三角形被存储为当前的包，并且type标记做相应的修改。此外，我们希望将该包存为一个逆时针次序的点集。这样就要求共线检测做一点修改，而不仅仅确定输入点是否位于或者不位于这条直线上。如果包为线段$\langle Q_0, Q_1\rangle$，并且输入点为P，图 13.35 显示了P与这条线段的 5 种可能的关系。

共线检测利用包含线段$\langle Q_0, Q_1\rangle$的直线的法线向量。如果$\vec{d} = Q_1 - Q_0$，那么$\vec{n} = -\vec{d}^{\perp}$，这是一个法线向量，当沿着线段从$Q_0$到$Q_1$时，该法线向量指向左边。定义$\vec{a} = P - Q_0$。图 13.35 所标识的 5 种可能性可用数学公式表示如下：

a. $\vec{n} \cdot \vec{a} > 0$

b. $\vec{n} \cdot \vec{a} < 0$

c. $\vec{n} \cdot \vec{a} = 0$ 且 $\vec{d} \cdot \vec{a} < 0$

d. $\vec{n} \cdot \vec{a} = 0$ 且 $\vec{d} \cdot \vec{a} > \vec{d} \cdot \vec{d}$

e. $\vec{n} \cdot \vec{a} = 0$ 且 $0 \leqslant \vec{d} \cdot \vec{a} \leqslant \vec{d} \cdot \vec{d}$

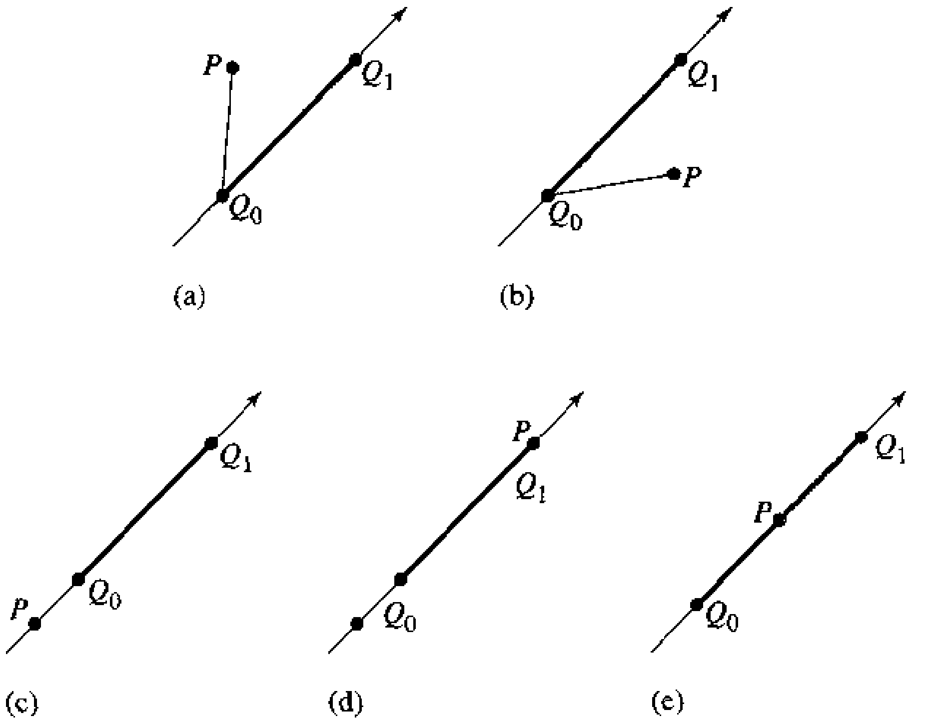


图 13.35　P 与端点为 Q_0 和 Q_1 的线段的 5 种可能的关系：（a）P 在线段的左边；（b）P 在线段的右边；（c）P 在线段所在的直线上，并在线段的左边；（d）P 在线段所在的直线上，并在线段的右边；（e）P 在线段所在的直线上，并在线段内

伪码使用一个整数标记来区分这些情形，按上面列出的测试次序排列的标记的值分别为 POSITIVE，NEGATIVE，COLLINEAR_LEFT，COLLINEAR_RIGHT 和 COLLINEAR_CONTAIN。

```
int CollinearTest(Point P, Point Q0, Point Q1)
{
    Point D = Q1 - Q0, N = -Perp(D), A = P - Q0;

    float NdA = Dot(N, A);
    if (NdA > 0)
        return POSITIVE;
    if (NdA < 0)
        return NEGATIVE;

    float DdA = Dot(D, A);
    if (DdA < 0)
        return COLLINEAR_LEFT;
    if (DdA > Dot(D, D))
        return COLLINEAR_RIGHT;

    return COLLINEAR_CONTAIN;
}
```

可以注意到，这 5 种可能性与当前包的顶点和包对输入点的可见性的讨论完全匹配。处理合并的伪码列出如下。如果包成为一个三角形，那么其顶点将按逆时针方向排序。

```
void MergeLinear(Point P, ConvexPolygon& hull, int& type)
{
    switch (CollinearTest(P, hull[0], hull[1])) {
        case POSITIVE:
            type = PLANAR;
            hull = {P, hull[0], hull[1]};
            break;
        case NEGATIVE:
            type = PLANAR;
            hull = {P, hull[1], hull[0]};
            break;
        case COLLINEAR_LEFT:
            // collinear order <P, Q0, Q1>
            hull = {P, Q1};
            break;
        case COLLINEAR_RIGHT:
            // collinear order <Q0, Q1, P>
            hull = {Q0, P};
            break;
        case COLLINEAR_CONTAIN:
            // collinear order <Q0, P, Q1>, hull does not change
            break;
    }
}
```

虽然在理论上正确无误，但是常见的浮点舍入误差将影响函数CollinearTest。检测的点可能只是接近共线的，但是却可能被当成共线来处理。一个健壮的系统可能更愿意利用模糊算术来进行共线检测。应该使用相对误差容许范围，以避免依赖于输入点的数值。一种可能的方法是对 $\vec{d}$ 和 $\vec{a}$ 之间的角度 θ 的余弦使用一个误差阈值 $\varepsilon > 0$。如果 $|\cos(\theta)| \leqslant \varepsilon$，那么 θ 可能被当成零来处理。即，如果 $|\vec{n}\cdot\vec{a}| = \|\vec{n}\|\|\vec{a}\||\cos(\theta)| \leqslant \varepsilon\|\vec{n}\|\|\vec{a}\|$，那么这三个点被当成共线来处理。在这个公式中，必须计算两个向量的长度，这是影响性能的一个问题。为了避免平方根运算，应该考虑平方后的方程，即 $|\vec{n}\cdot\vec{a}|^2 \leqslant \varepsilon\|\vec{n}\|^2\|\vec{a}\|^2$，我们在此使用 ε 作为 $|\cos(\theta)|^2$ 的允许范围。对于三个点共线的情形，可以使用一种相似的误差允许范围。包含性的参数区间为 $[0, \|\vec{d}\|^2]$，但是可以被扩展为 $[-\varepsilon\|\vec{d}\|^2, (1+\varepsilon)\|\vec{d}\|^2]$。在下面列出的处理这种情形的伪码中，epsilon0和epsilon1的合适值由程序的编写者来确定。

```
int CollinearTest(Point P, Point Q0, Point Q1)
{
    Point D = Q1 - Q0, A = P - Q0;
    float NdA = D.x * A.y - D.y * A.x;  // N = -Perp(D) = (-D.y, D.x)
    float NdN = D.x * D.x + D.y * D.y;  // |N| = |D|
    float AdA = A.x * A.x + A.y * A.y;

    if (NdA * NdA > epsilon0 * NdN * AdA) {
        if (NdA > 0)
            return POSITIVE;
        if (NdA < 0)
            return NEGATIVE;
    }
```

```
    float DdA = Dot(D, A);
    if (DdA < -epsilon1 * NdN)
        return COLLINEAR_LEFT;
    if (DdA > (1 + epsilon1) * NdN)
        return COLLINEAR_RIGHT;

    return COLLINEAR_CONTAIN;
}
```

一旦当前的包中具有三个或者更多的点，不管后面的输入点的值是什么，都可以保证保持一个具有正面积的凸多边形。

要讨论的最后一个函数是MergePlanar。如果存在三角形，那么一旦第一个三角形被MergeLinear建立，则得到这个三角形的最后一个插入点总是存储在hull[0]中。该点是搜索下一个输入点和当前包的切点的一个合适的候选点。实现平面合并的代码包含两个循环，一个用于寻找上面的切点，另一个用于寻找下面的切点。循环体只是检测基于将Collinear应用于输入点和当前包的边所得的结果的可见性。伪码为

```
void MergePlanar(Point P, ConvexPolygon& hull)
{
    // find upper tangent point
    for (U = 0; i = 1; U < hull.N; U = i, i = (i + 1) mod hull.N) {
        test = CollinearTest(P, hull[U], hull[i]);

        if (test == NEGATIVE)  // edge visible, go to next edge
            continue;

        if (test == POSITIVE            // edge not visible,
        ||  test == COLLINEAR_LEFT) {  // only edge end point is visible
            // upper found
            break;
        }

        // test == COLLINEAR_CONTAIN || test == COLLINEAR_RIGHT
        // Theoretically cannot occur when input points are distinct and
        // sorted, but can occur because of floating-point round-off
        // when P is very close to the current hull.  Assume P is on the
        // hull polygon--nothing to do.
        return;
    }

    // find lower tangent point
    for (L = 0; i = hull.N - 1; i >= 0; L = i, i--) {
        test = CollinearTest(P, hull[i], hull[L]);

        if (test == NEGATIVE)  // edge visible, go to next edge
            continue;

        if (test == POSITIVE             // edge not visible,
        ||  test == COLLINEAR_RIGHT) {  // only edge end point is visible
            // lower found
```

```
                break;
            }
            // test == COLLINEAR_CONTAIN || test == COLLINEAR_LEFT
            // Theoretically cannot occur when input points are distinct and
            // sorted, but can occur because of floating-point round-off
            // when P is very close to the current hull.  Assume P is on the
            // hull polygon--nothing to do.
            return;
        }

        // Both tangent points found.  Now do:
        // 1. Remove visible edges from current hull.
        // 2. Add new edges formed by P and tangent points.
    }
```

伪码中包含的第一步和第二步中更新包的最简单的算法是用迭代法从当前包和输入点中建立一个临时包：

```
ConvexPolygon tmpHull;
tmpHull[0] = P;
for (i = 1; true; i++, U = (U + 1) mod hull.N) {
    tmpHull[i] = hull[U];
    if (U == L)
        break;
}
hull = tmpHull;
```

然而，迭代是$O(n)$级的，因此，增量算法成为$O(n^2)$级的。为了避免这一点，将包维护成某种连接在一起的结构类型，使得可见的边的连接链可以在切点断开（这是一种$O(1)$级运算，而且删除需要在$O(1)$级时间内完成），这是非常重要的。该链不应该一次删除一个节点，否则，将回到$O(n)$级时间。这就要求一种密切关注内存管理的实现。在连在一起的链被删除以后，新的连接被加入到从表示P的节点到表示切点的节点链中。

有了合适的算法结构后，还应该注意一个重要的方面。由浮点舍入误差所产生的所有问题都封装在函数CollinearTest内。使用增量包算法的应用程序的任何意外结果都只能是由CollinearTest的实现产生的，特别是在选择相对误差阈值ε_0和ε_1时所产生的。这种封装使得调试应用程序所产生的问题很容易。

2. 分而治之算法

分而治之算法（divide-and-conquer method）是计算机科学中的一个标准范例。其思想是，对于一个问题，将其分解为相同类型的两个较小的问题，解决这两个较小的问题，将得到的结果合并在一起，以构成原来问题的解。如果一个问题有n个输入，求解该问题所需的时间为T_n，将其分解为两个较小的问题，每一个问题各具有一半的输入，这样可得到递归公式$T_n = 2T_{n/2} + M_n$。每一个较小的问题都有（近似于）$n/2$的输入，并需用$T_{n/2}$时间来求解（根据T_k的定义）。数量M_n表示合并两个较小问题的解所需的时间。如果M_n需要线性时间，即$O(n)$，那么可以证明求解所需的时间为$O(n \log n)$。Cormen，Leiserson 和 Rivest（1990）详细讨论了这种形式的递归。将分而治之方法应用于凸包，其合并时间是线性的，

因此对于n个点集的凸包算法，需要的时间为$O(n\log n)$。

与凸包的增量构建一样，输入点按相同的方式排序：当$x_0 < x_1$或者当$x_0 = x_1$且$y_0 < y_1$时，$(x_0, y_0) < (x_1, y_1)$。我们也不设定点集的结构。与在增量凸包算法中一样，输入点被排序，重复点被删除。初始的伪码为

```
ConvexPolygon DividAndConquerHull(int n, Point V[n])
{
    Sort(n, V);
    RemoveDuplicates(n, V);  // n can decrease, V has contiguous elements
    ConvexPolygon hull;
    GetHull(0, n - 1, V, hull);
    return hull;
}
```

递归构建出现在GetHull中，在下面显示其结构。值i0和i1分别是点的子集的第一个和最后一个索引，必须计算这个子集的凸包。

```
void GetHull(int i0, int i1, Point V[], ConvexPolygon& hull)
{
    int quantity = i1 - i0 + 1;
    if (quantity > 1) {
        // middle index of input range
        int mid = (i0 + i1) / 2;

        // find hull of subsets (mid - i0 + 1 >= i1 - mid)
        ConvexPolygon LHull, RHull;
        GetHull(i0, mid, V, LHull);
        GetHull(mid + 1, i1, V, RHull);

        // merge the convex hulls into a single convex hull
        Merge(LHull, RHull, hull);
    } else {
        // convex hull is a single point
        hull[0] = V[i0];
    }
}
```

当然，其中的技术问题是函数Merge如何计算两个凸包所组成的凸包。其思想是计算一个点、一个凸集和一个凸多边形的凸包的思想的扩展。在后一种情形中，合并取决于寻找凸包的上面和下面的包含一个点的切线。图 13.34 显示了这种典型的情形。对于两个凸多边形，我们仍然需要寻找两个凸包的上面和下面的切线。图 13.36 显示了这种典型的情形。与增量凸包问题不同，在开始时我们并不知道一个切点。分别对每一个多边形进行对点对的完整搜索，将导致合并的时间为$O(n^2)$级。如果我们对左包H_L和右包H_R使用如下的行走算法，那么速度更快，所需时间为$O(n)$级。

这种方法用于寻找包下面的切线。在H_L上寻找具有最大的x分量的点P_i，这是一种线性时间级的处理。在H_R上寻找具有最小的x分量的点Q_j，这也是一种线性时间级的处理。如果包含P_i和Q_j的直线不是H_L的切线，则按顺时针方向遍历H_L的顶点（减小i），直到直线成为H_L的切线。现在切换到H_R。如果包含当前P_i和Q_j的直线不是H_R的切线，则按逆时

针方向遍历H_R的顶点（增加j），直到直线成为H_R的切线。这种遍历产生一条新的直线，该直线可能不再与H_L相切。交替地重复遍历，直到这条直线与两个包都相切。图 13.37 显示了一种典型的情形。其极值点为P_0和Q_2。标记为 1 的初始线段为$\langle P_0, Q_2\rangle$。该线段不与左包相切于P_0，因此左包的索引被减小（与 5 的模），标记为 2 的新线段为$\langle P_4, Q_2\rangle$。这条线段与左包相切于P_4，但是并不与右包相切于Q_2。右包的索引增加，而且标记为 3 的新线段为$\langle P_4, Q_3\rangle$。这条线段与右包不相切，因此右包的索引再增加，并且标记为 4 的新线段为$\langle P_4, Q_4\rangle$。这条线段与右包相切，但是与左包并不相切。左包索引减小，而且标记为 5 的新线段为$\langle P_3, Q_4\rangle$。这条线段与两个包都相切，并且是包下面的切线。切线可能与包的一条共用切点的边共线，但是与其对每一次合并都进行这样的测试并扩展切线以包括共线的边，还不如在建立包时，在对所返回的凸多边形的后加工处理中删除共线的边。

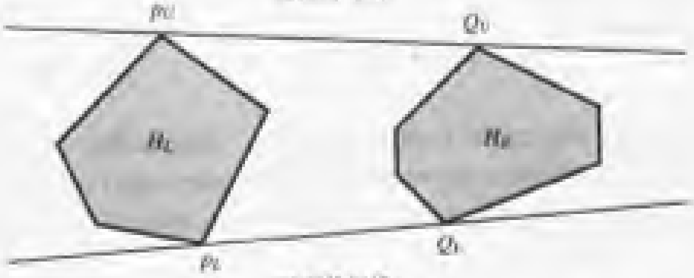


图 13.36　两个凸包H_L和H_R及其上下方的切线

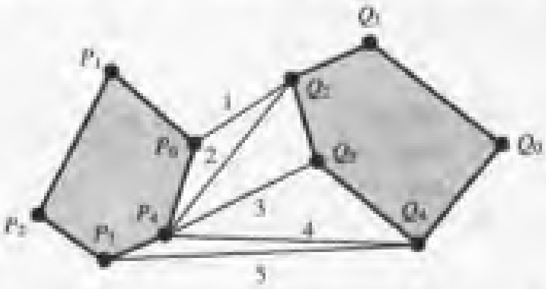


图 13.37　两个凸包H_L和H_R及寻找较低的切线的增量式搜索

下面列出了Merge的伪码。其中对接近递归生成的隐含的二叉树的叶子节点的较小的集合进行了特殊的处理。

```
void Merge(ConvexPolygon LHull, ConvexPolygon RHull, ConvexPolygon& hull)
{
    if (LHull.n == 1 && RHull.n == 1) {
        // duplicate points were removed earlier, the hull is a line segment
        hull[0] = LHull[0];
        hull[1] = RHull[0];
        return;
    }
    if (LHull.n == 1 && RHull.n == 2) {
        // merge point and line segment, result in RHull
        MergeLinear(LHull[0], RHull);
        hull = RHull;
```

```
        return;
    }
    if (LHull.n == 2 && RHull.n == 1) {
        // merge point and line segment, result in LHull
        MergeLinear(RHull[0], LHull);
        hull = LHull;
        return;
    }
    if (LHull.n == 2 && RHull.n == 2) {
        // merge point and line segment, result in LHull
        MergeLinear(RHull[1], LHull);
        if (LHull.n == 2) {
            // RHull[1] was on line of LHull, merge next point
            MergeLinear(RHull[0], LHull);
            hull = LHull;
            return;
        }

        // RHull[1] and LHull form a triangle.  Remove RHull[1] so that
        // RHull is a single point.  LHull has been modified to be a
        // triangle.  Let the tangent search take care of the merge.
        RHull.Remove(1);
    }

    // find indices of extreme points with respect to x
    LMax = IndexOfMaximum{LHull[i].x};
    RMin = IndexOfMinimum{RHull[i].x};

    // get lower tangent to hulls, start search at extreme points
    LLIndex = LMax;  // lower left index
    LRIndex = RMin;  // lower right index
    GetTangent(LHull, RHull, LLIndex, LRIndex);

    // get upper tangent to hulls, start search at extreme points
    ULIndex = LMax;  // upper left index
    URIndex = RMin;  // upper right index
    GetTangent(RHull, LHull, URIndex, ULIndex);

    // construct the counterclockwise-ordered merged-hull vertices
    ConvexPolygon tmpHull;
    i = 0;
    for (each j between LRIndex and URIndex inclusive) {
        tmpHull[i] = hull[j];
        i++;
       }
       for (each j between ULIndex and LLIndex inclusive) {
           tmpHull[i] = hull[j];
           i++;
       }
       hull = tmpHull;
}
```

函数MergeLinear与用于增量凸包建立的同名函数具有相同的结构，只是在语义上有很少的修改，即输入的线段多边形被改为用于存储合并的包。

切线搜索使用与增量包构建中的可见性一样的概念。正如在前面所介绍的，在两个输入包之间连续搜索。输入的索引L和U是开始搜索的索引。在返回时，索引对应于在两个凸包上的切点。

```
void GetTangent(ConvexPolygon LHull, ConvexPolygon RHull, int& L, int& R)
{
    // In theory the loop terminates in a finite number of steps, but the
    // upper bound for the loop variable is used to trap problems caused by
    // floating-point round-off errors that might lead to an infinite loop.

    for (int i = 0; i < LHull.n + RHull.n; i++) {
        // end points of potential tangent
        Point L1 = LHull[L];
        Point R0 = RHull[R];

        // walk clockwise along left hull to find tangency
        int Lm1 = (L - 1) mod LHull.n;
        Point L0 = LHull[Lm1];
        int test = CollinearTest(R0, L0, L1);
        if (test == NEGATIVE || test == COLLINEAR_LEFT) {
            L = Lm1;
            continue;
        }

        // walk counterclockwise along right hull to find tangency
        int Rp1 = (R + 1) mod RHull.n;
        Point R1 = RHull[Rp1];
        test = CollinearTest(L1, R0, R1);
        if (test == NEGATIVE || test == COLLINEAR_RIGHT) {
            R = Rp1;
            continue;
        }
        // tangent segment has been found
        break;
    }

    // Trap any problems due to floating-point round-off errors.
    assert(i < LHull.n + RHull.n);
}
```

由于每一个顶点都位于一个包上，使用CollinearTest来检测位于另一个包上的当前边的可见性。当可见时，返回值一般是NEGATIVE。然而，当包上的初始极值点位于相同的垂直直线上时，必须小心处理。图 13.38 显示了一种典型的情形。初始的候选切线是$\langle L_1, R_0\rangle$，如图 13.38（a）所示。在试图遍历左包时，CollinearTest(R0,L0,L1)的输出是 COLLINEAR_RIGHT。左包的索引保持不变，并且试图遍历右包CollinearTest(L1,R0,R1) 的输出也是 COLLINEAR_RIGHT。在这种情形中，右包的索引增加，就好像R_0位于共用的垂直直线的右边一点。图

13.38（b）显示了增量后的当前状态。遍历切换回左包。CollinearTest(R0,L0,L1)的输出还是COLLINEAR_RIGHT，而左包的索引保持不变。遍历切换回右包。CollinearTest(L1,R0,R1) 的输出是COLLINEAR_RIGHT，而右包的索引增加，就好像R_0位于共用的垂直直线的右边一点。图13.38（c）显示了增量后的当前状态。切换回左包。CollinearTest(R0,L0,L1)的输出是NEGATIVE，因为边$\langle R_0, R_1\rangle$是完全可见的。左包的索引增加。遍历切换回右包。图13.38（d）显示了减小后的当前状态。循环再迭代一次，然而CollinearTest的两次调用都返回POSITIVE，而且图13.38（c）中的$\langle L_1, R_0\rangle$与两个包相切。

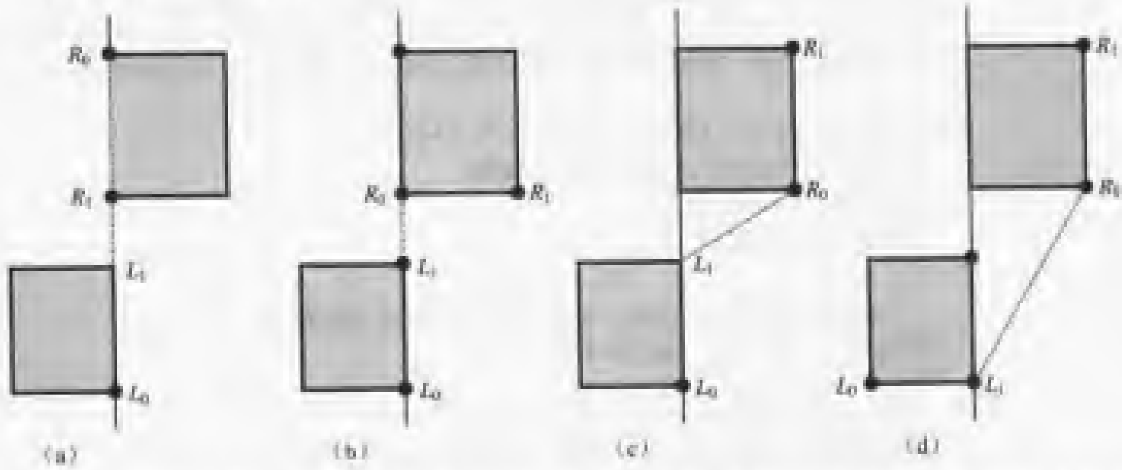


图 13.38　用于初始化切线搜索的极端点位于同一条垂直线上。虽然初始的可见性测试都不会产生否定结果，但连接极端点的初始线段并不是包的切线。当前的候选切线用点线显示

一旦找到了包的两条切线，那么合并包的构建就与增量包的构建在结构上完全一样。伪码显示了包含取自基于切点位置的两个包的索引的子集的临时凸多边形的创建，但是与增量构建一样，可以使用一个连接的表结构，并且，分离、附加和子表删除都可在$O(1)$时间内完成。然而，总共的时间还是$O(n)$，因为寻找极值点要遍历两个输入包。

13.7.2　三维凸包

在二维空间中使用增量方法或者分而治之方法来构建点集的凸包的思想可以自然地扩展到三维空间。增量方法的扩展可以很简单地实现。实现分而治之方法则要复杂得多。

1. 增量构建算法

三维空间中的算法与二维空间中的算法相似。每一个点都会被处理并与前一个点的凸包合并。监测包的维数，以确保共线的点产生线段包，而共面的点产生平面凸多边形。典型的情形是，包成为凸多面体。

合并操作比二维空间的情形稍微复杂一些。我们不再获得两条切线，而是获得一个可见的锥体（不要与作为二次曲面的圆锥面相混淆），其顶点就是要合并的点，其终端边构成一个封闭的折线，折线的每一条线段都是当前包的分离可见面与不可见面的边。封闭的折线有时称为明暗界限，这是一个用于天文学的术语，表示天体明亮和阴暗区域的边界。图13.39显示了一种典型的情形。可见的面必须从当前包中删除，而必须增加可见锥体的面。实现这一操作的一种简单的算法涉及遍历所有的当前面，寻找有一条边位于明暗界限

上的面，并在某种数据结构中存储这些边。再遍历时，可见面被排除，而隐藏面被保留。一旦访问了所有的面，就得到了明暗界限的封闭折线。折线被遍历，构建由每一条经过P点的边所构成的面，并将其增加到合并包的数据结构中。

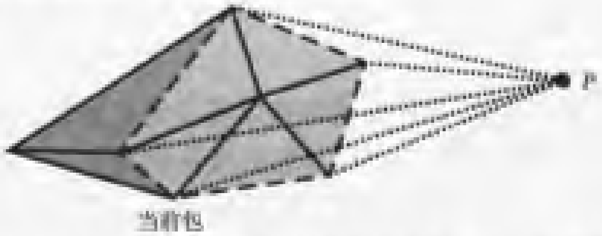


图 13.39 当前包和点将合并在一起。可见的面用浅灰色表示。隐藏面用深灰色表示。分开这两个集合的折线用虚线显示。可见锥体的其他边用点线表示

有一种效率更高的寻找明暗界限的算法，但是其实现更加复杂，它使用了对明暗界限的边的线性搜索。如果一个面位于平面$\hat{n}\cdot X+d=0$（其中$\hat{n}$是一条单位长度法线）上，点P与平面之间的有符号距离为$\delta=\hat{n}\cdot P+d$。如果$\delta>0$，那么平面对P是可见的（因此对应的面也是可见的）。如果$\delta\leqslant 0$，那么平面对P是隐藏的，对应的面也是隐藏的。在$\delta=0$的情形中，与这个面最接近的平面的边可能是对P可见的，但是这条边是$\delta\geqslant 0$时的另一个面的一部分。当前凸包的面网格具有对偶图，其节点表示面，而其弧表示面的边。特别地，两个节点之间的弧说明对应的面是相邻的。每一个节点都赋予P与面之间的有符号距离值。从图的一个具有正的有符号距离值的节点开始搜索，以寻找距离减小（更精确地说，是不增加）的节点的路径。从当前节点要访问的下一个节点就是在所有相邻的节点中有符号距离为最小正值的节点。由于包是凸的，因此最终必须到达一个节点，它的相邻节点中至少有一个具有非正的有符号距离值。根据定义，共享的边位于明暗界限上。一旦找到明暗界限的第一条边，并假设一种维护每一个顶点的相邻边列表的数据结构，就可以遍历明暗界限。

实际上，这种方法与寻找一种基于图的像素来定义的图像（有符号距离）的零级曲线密切相关。算法的次序可直接据此确定。如果图像是n个像素的方块（在我们的问题中是面的数量），寻找在零级曲线上的第一个点的线性搜索就是$O(\sqrt{n})$级的。沿着零级曲线的遍历（在我们的问题中是明暗界限）也是线性搜索，也需要$O(\sqrt{n})$级的时间。前面提到的简单算法访问所有的三角形，需要$O(n)$级时间，是一种渐近较慢的算法。

顶级调用的伪码为

```
ConvexPolyhedron IncrementalHull(int n, Point V[n])
{
    Sort(n, V);
    RemoveDuplicates(n, V);
    ConvexPolyhedron hull;
    type = POINT;
    hull[0] = V[0];
    for (i = 1; i < n; i++) {
        switch (type) {
            case POINT:   type = LINEAR; hull[1] = V[i]; break;
```

```
                case LINEAR:  MergeLinear(V[i], hull,type);    break;
                case PLANAR:  MergePlanar(V[i], hull,type);    break;
                case SPATIAL: MergeSpatial(V[i], hull);        break;
            }
        }
        return hull;
    }
```

表示凸多面体的空间特性的数据结构与表示其他特性的数据结构很不相同。一个线性包的自然存储方式是两个点的数组，即作为包的线段的端点。一个平面包的自然存储方式是有序点的数组或列表。一个空间包的自然存储方式更加复杂。在最抽象的形式中，其数据结构是一个顶点—边—面表，该表允许增加和删除每一个基本元素。在一个基于三角形的应用程序中，面被存储为三角扇形。对于二维的凸多边形，可以增加对共线的边收缩成一条边的支持。在三维空间中，三角扇形可以收缩成凸多边形，而这些凸多边形的共线的边可以收缩成一条边。

函数MergeLinear几乎与用于二维空间增量包的同名函数完全相同。然而，如果三个输入点（下一个将要合并的点和当前线段包的两个端点）是不共线的，那么它们位于同一个平面上，并且在选择平面的一条法线之前，它们并没有指定的次序（即二维情形中的正或负）。应该选择一条法线，从而如果包最终称为空间包，则第一个面是一个凸多边形，而法线可用于对顶点重新排序（如果必要的话），这样，当从包外观察时，多边形是逆时针排序的。

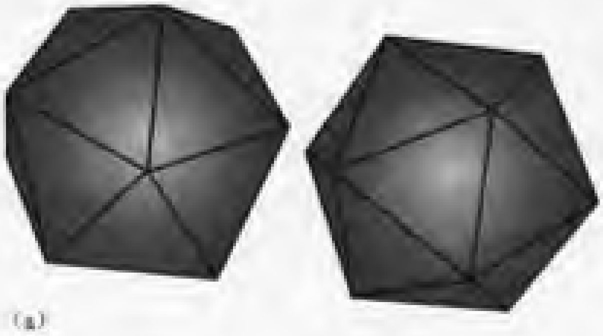
函数MergePlanar与二维情形中的同名函数有一点不同。如果下一个输入点位于当前平面上，那么应用二维空间合并算法来将当前的包（一个凸平面多边形）更新为另一个凸平面多边形。当然，将点作为三维对象来合并。如果下一个输入点不位于当前平面上，那么这个包就成为空间包，并使用MergeSpatial来进行后面的合并。如果用于表示凸多面体的数据结构是一个存储为顶点—边—面表的三角形网格，那么当前的包，即一个凸平面多边形，必须被扇形分解成三角形，并加入该表中。由下一个输入点和凸多边形的边所构成的另一个三角形也被加入到这个表中。这时可以利用前面计算得到的法线向量，以保证当从空间包的外面观察时，加入的三角形是逆时针排序的。

函数MergeSpatial执行前面描述的功能。无论如何，当前包的可见面都被删除，而增加由明暗界限和下一个输入点所构成的新面。

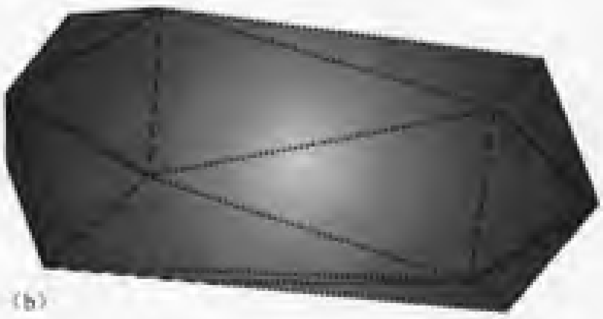
2. 分而治之算法

这种算法的基本架构与二维情形相似。输入的点沿某些轴排序。点集被划分为两个集合，对这些点集进行递归计算以得到包。用一个$O(n)$时间级的算法将计算得到的包合并成一个包。

这种算法的思想是用一个平面将两个输入包包围起来。最终的结果是由三角形和（或者）四边形所组成的条，这些三角形和四边形就成为合并包的新面。图 13.40 说明了将两个二十面体合并成一个凸多面体的情形。包围开始于寻找针对这两个输入包的一个平面，这个平面与两个包都相切。对每一个包，切点集是一个顶点、一条边或者一个面。如果这个集合是一个面，我们就仅需这个面的一条边（这条边对其他的支持点集来说是可见的）来开始包围处理。由于我们仅需要考虑顶点和边，因此位于合并包上的新面不是三角形就是四边形。



（a）



（b）

图 13.40　（a）两个二十面体。（b）合并的包。虚线标识部分源包的面的边。点线表示部分新增的面的边

不论这些包所支持的点集的类型是什么，都必定存在顶点P_0和Q_0，各属于一个包，它们组成的线段$\langle P_0, Q_0\rangle$是合并包的一条边。包含这条边的支持平面的一半沿着包含这条边的直线"卷起来"，直到遇到包的另一个顶点为止。如果这个顶点位于第一个包上，称之为P_1，那么它必定与P_0相邻。三角形$\langle P_0, P_1, Q_0\rangle$是合并包的一个面。类似地，如果这个顶点位于第二个包上，称之为Q_1，那么它必定与Q_0相邻。三角形$\langle Q_0, Q_1, P\rangle$是合并包的一个面。也可能同时遇到$P_1$和$Q_1$，在这种情形中，这 4 个点所构成的四边形是合并包的一个面。平面也沿着包含最后找到的面的第一条边的直线卷起来。重复这种处理直到遇到原始的卷边为止。正如 O'Rourke（1998）所描述的，渐近分析说明分摊的时间为$O(n)$级。

一旦通过平面包围构建了合并包的面，就必须删除不再可见的老的面。在三维增量包的构建中，对一个点或者凸多面体进行合并。我们知道，明暗界限是位于凸多面体上的边的封闭折线，它分离可见面和与一个点相关的隐藏面。作为一个图，它的节点为明暗界限的顶点并且它的弧为连接连续的顶点的边，明暗界限就是一个简单的圆。增量包的合并步骤与寻找明暗界限相关。最有效的算法是寻找明暗界限的一条边，然后遍历分开两个面的明暗界限的相邻的边，其中一个面具有正的有符号距离，另一个面具有非正的有符号距离。由于这种明暗界限是一个简单的圆，因此这种遍历能成功进行。图 13.40 可能使你相信，这两个输入包的明暗界限都是简单的圆。然而，实际上，这种情形并不是必需的。说明这种情形的一个简单的例子涉及合并一个凸多面体和一条线段。保持为合并包的面就是对所有位于线段上的点都隐藏的面。相应地，被抛弃的面就是对某些位于线段上的点可见的面。图 13.41 显示了一个例子，其中的凸多面体是一个棱锥。这个棱锥的明暗界限由共用一个顶点的两个三角形组成。由于存在上述的可能性，当试图删除每一个输入包的可见面时必须小心，不要假设明暗界限是一个简单的圆。正确的方法是，遍历明暗界限的所有边，并

分离可见面和隐藏面。确实，位于一个输入包上的可见面都位于同一个互连的对象上，因此，可以使用一种深度优先的搜索来一次一个地删除它们。然而，只要应用程序提供一种切实可行的内存管理方法，用以管理在二维增量包构建中所提到的直线，那么只需进行一点额外的工作，就可以使用一个$O(1)$级的删除算法。其思想是，分离可见面和隐藏面，但是允许可见面的分量存在，直到完整的包构建完成为止。前面的拷贝和所有的变化分量都是内存中的临时工作区的一部分，将立即被删除。

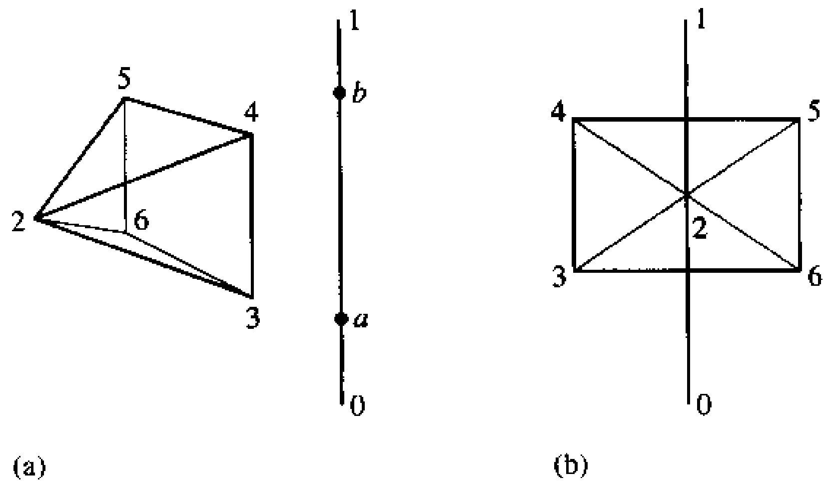


图 13.41　(a) 棱椎和线段的侧视图；(b) 从线段后面观察的视图。线段$(0,a)$只能看见三角形$(2,3,6)$和四边形$(3,4,5,6)$。线段(a,b)只能看见四边形。线段$(b,1)$只能看见三角形$(2,4,5)$和四边形。在上述各种情形中都隐藏的面是三角形$(2,3,4)$和$(2,5,6)$。包含这些三角形的边、一组线段的最终图形是两个圆，而不是一个简单的圆

13.7.3 高维凸包

高维凸包算法的实现更加复杂，但是增量构建和分而治之构建都可以自然地扩展到高维凸包。Klee（1980）证明了d维空间中的n个点的凸包至少可以有$n^{\lfloor d/2 \rfloor}$个超面，因此渐近方法的效果更差。特别地，在维数$d=4$时，超面的数量可以是二次方的，因此不可能建立一个$O(n\log n)$级算法。

1. 增量构建算法

当维数h从0（一个点）开始并增加到一个最终的值$h\leqslant d$（其中d是整个空间的维数）时，都维护当前的包。对当前包维护一个原点和一个正交集基向量。属于输入集的初始点保存为原点A。每一次插入一个输入点都迫使h增加，并加入一个与前面的所有基向量都正交的单位长度基向量$\hat{d}_h$。对输入点的第二个拷贝排序，以此为代价，可以进行支持点集的排序而不影响原来的集合的某些操作，但这样的操作在低维时可以高效地进行，原点和基可用于表示坐标系中已经处理的点。其优点是，只要将点表示为数组，一种针对一个特定维的凸包方法就可被用于高维的凸包方法重用。特定维的点坐标存储在数组的连续位置上，因此凸包方法可以安全地在数组内访问这些坐标。最困难的技术挑战可能就是维护在每一维内凸包的数据结构。在三维空间中，一个顶点一边一三角形表可用于存储这个包（多余三个顶点的凸多边形的面被三角扇形化）。在四维空间中，要求有一个顶点一边一三角

形一四面体表。随着维数的增加，数据结构的复杂度也增加。

这里提供一个更高层的概要说明，假设至少有两个不同的点，P_0和P_1。初始时，点被存储。初始点被存储为原点$A = P_0$。初始基底向量为$\hat{d}_1 = (P_1 - P_0)/\|P_1 - P_0\|$。当前包的维数为$h = 1$。对于每一个维度$h \leqslant 4$，合并函数表示为`Merge<h>`。表示包的数据结构表示为`Hull<h>`。数据结构中的每一个点都存储为一个四维数组，但是仅有开始的h个分量是相关的。因此，初始的线性包可以有效地仅存储为两个常数$s_i = \hat{d}_i \cdot P_i$，$0 \leqslant i \leqslant 1$，它们表示四维点$P_i = A + s_i\hat{d}_i$（因此$s_0 = 0$）。

线性合并算法可表示如下。

```
void Merge<1>(Point P, Hull<1> hull)
{
    // uses affine coordinate system {A; D1}
    B = P - A;
    t1 = Dot(D1, B);
    R = B - t1 * D1;  // project out D1 component
    if (|R| > 0) {
        // |R| is the length of R
        // dimension increases
        h = 2;
        D2 = R / |R|;  // affine coordinate system becomes {A; D1, D2}
        t2 = Dot(D2, B);
        ReformatAndInsert(hull, t1, t2); // P = A + t1 * D1 + t2 * D2
    } else {
        // hull is still linear
        Update(hull, t1);
    }
}
```

处理包的函数`ReformatAndInsert`并不复杂。包是一条线段，由一对点来表示。`(t1,t2)`的插入要求创建一个三角形。这个三角形可以存储为点的一个三元组。如果包保持为平面的，并且更多的点被合并，以构成一个多于三条边的凸多边形，那么这些点可以被当成一个有序列表来维护。数据结构`Hull<1>`表示具有常数s_0和s_1的线段的端点。在重新格式化时，这些值变成$(s_0, 0)$和$(s_1, 0)$，它们与(t_1, t_2)位于同一空间中。所有这样的两个向量都是原点A和基向量$\hat{d}_1$和$\hat{d}_2$所构成的仿射系统中的坐标。

函数`Update`寻找包的明暗界限，在这种情形中，就是作为包的线段的一个端点。由于已排序，这些端点中的一个对输入点P肯定是可见的。线段被适当地更新，以替换从P点可见的端点。在一维空间中进行这种计算。由于`hull`表示为`Hull<1>`的形式，只有第一个数组项是相关的，所有的方法都可以访问它。

平面合并的代码可以表示如下。

```
void Merge<2>(Point P, Hull<2> hull)
{
    // uses affine coordinate system {A; D1, D2}
    B = P - A;
    t1 = Dot(D1, B);
    t2 = Dot(D2, B);
```

```
        R = B - t1 * D1 - t2 * D2;  // project out D1, D2 components
        if (|R| > 0) {
            // dimension increases
            h = 3;
            D3 = R / |R|;  // affine coordinate system becomes {A; D1, D2, D3}
            t3 = Dot(D3, B);
            ReformatAndInsert(hull, t1, t2, t3);
            // P = A + t1 * D1 + t2 * D2 + t3 * D3
        } else {
            // hull is still planar
            Update(hull, t1, t2);
        }
    }
```

与前面介绍的维中的函数相比，处理包的函数ReformatAndInsert并不简单。包是表示为有序点列表的平面凸多边形。如果空间中的凸包表示法为顶点一边一面表，那么当前包被当成面加入该表中。(t1,t2,t3)的插入要求合并包的额外的三角形面，每一个这样的三角形面都由该点和凸多边形的一条边构成。如果包表示为顶点一边一三角形表，凸多边形必须首先被三角扇形化。三角形被加入到该表中。被插入的点和凸多边形的一条边所构成的三角形面被加入到该表中。数据结构 Hull<2> 将点表示为(s_1, s_2)。在重新格式化时，这些值成为$(s_1, s_2, 0)$，与(t_1, t_2, t_3)位于相同的空间中。所有这样的三个向量都是原点A和基向量$\hat{d}_1$、$\hat{d}_2$和$\hat{d}_3$所构成的仿射系统中的坐标。

函数Update寻找包的明暗界限，在这种情形中，就是包的两个切点，它们构成一个顶点为输入点的可见锥体。对P可见的边被删除，而由P和切点所构成的边被加入到包中。

空间合并的代码可以表示如下。

```
void Merge<3>(Point P, Hull<3> hull)
{
    // uses affine coordinate system {A; D1, D2, D3}
    B = P - A;
    t1 = Dot(D1, B);
    t2 = Dot(D2, B);
    t3 = Dot(D3, B);
    R = B - t1 * D1 - t2 * D2 - t3 * D3;
    // project out the D1, D2, D3 components
    if (|R| > 0) {
        // dimension increases
        h = 4;
        convert hull from Hull<3> format to Hull<4> format;
        D4 = R / |R|;
        // affine coordinate system becomes {A; D1, D2, D3, D4}
        t4 = Dot(D4,B);
        ReformatAndInsert(hull, t1, t2, t3, t4);
        // P = A + t1 * D1 + t2 * D2 + t3 * D3 + t4 * D4
    } else {
        // hull is still spatial
        Update(hull, t1, t2, t3);
    }
}
```

处理包的函数ReformatAndInsert也不复杂。三维包是一个凸多面体，为了便于讨论，将其存储为顶点一边一三角形表。同时，也是为了便于讨论，假设四维包存储为顶点一边一三角形一四面体表。凸多面体必须首先被分区为四面体，这是一种比三角扇形化稍微复杂一点的处理。四面体被加入到表中。由插入点和凸多面体的三角形所构成的四面体的面被加入到表中。数据结构 Hull<3> 将点表示为(s_1, s_2, s_3)。在重新格式化时，这些值变成与(t_1, t_2, t_3, t_4)位于同一空间的$(s_1, s_2, s_3, 0)$。所有这样的 4 个向量都是原点A和基向量$\hat{d}_1$, $\hat{d}_2$, $\hat{d}_3$和$\hat{d}_4$所构成的仿射系统中的坐标。

函数Update寻找凸包的明暗界限，在这种情形中，就是分离可见面与隐藏面的简单的封闭折线。对P可见的面被删除，而由P和明暗界限的边所构成的面被加入到包中。

最后，下面显示了超空间合并的算法。由于源空间的维数为 4，不需要将基底分量投影，因为对应的向量$\vec{r}$（即伪码中的R）将总是为$\vec{0}$。

```
void Merge<4>(Point P, Hull<4> hull)
{
    // Uses affine coordinate system {A; D1, D2, D3}.
    // Hull remains hyperspatial.
    B = P - A;
    t1 = Dot(D1, B);
    t2 = Dot(D2, B);
    t3 = Dot(D3, B);
    t4 = Dot(D4, B);

    Update(hull, t1, t2, t3, t4);
}
```

函数Update寻找凸包的明暗界限，即分离可见的超面与不可见的超面的面的集合。对P可见的超面被删除，由P和明暗界限的面所构成的新的超面被加入到包中。由于最后的过程处理 4 个向量，并不需要仿射坐标，并且可在原来的坐标系中处理点。如果在实现中确定这种选择，那么可以扩展处理维数增加到 4 的Merge<3>中的代码段，用原来的点替换当前包中的（存储在仿射空间中的）所有点。这就要求保存输入点的原始索引，但是，这是在实现低维算法时的方式，因为输入的数据集的索引应该用于存储互连信息，而不是已排序数据集的索引。如果做了选择，那么Merge<4>也仅需调用更新函数。

如果通过投影，一个合并函数被保持为一般形式，那么我们可以实现一个单一的、通用的合并函数，如下所示。

```
void Merge<k>(Point P, Hull<k> hull)
{
    B = P - A;
    R = B;
    for (i = 1; i <= k; i++) {
        t[i] = Dot(D[i], B);
        R = R - t[i] * D[i];
    }

    if (|R| > 0) {
        // dimension increases
```

```
            h = h + 1;
            convert hull from Hull<k> format to Hull<k + 1> format;
            D[k + 1] = R / |R|;
            t[k + 1] = Dot(D[k + 1], B);
            ReformatAndInsert<k + 1>(hull,t );  // t = (t[1],...,t[k + 1])
        } else {
            // hull remains the same dimension
            Update<k>(hull, t);  // t = (t[1],...,t[k])
        }
    }
```

为了保持实现的简单化，函数Update<k>可以写成只是对超面进行迭代，并且保留隐藏面和删除可见面，同时跟踪可见超面和不可见超面所共用的每一个面，那么对这些面进行迭代，并与P一起构成超面，然后将它们加入以形成合并包。

这种方法中的浮点数问题主要表现在如下三个方面。首先，仿射坐标t[i]的计算将与浮点舍入误差有关。其次，$\vec{r}$的长度与零的比较应该用其与一个很小的数的比较来代替，即一个正的阈值。即使如此，这样的比较还是会受到输入点的值的影响。第三，函数Update检测超面的可见性也与浮点数相关。每一次检测都是一个$(k+1)\times(k+1)$的行列式计算，其中矩阵的第一行的所有元素都是 1，而不包括第一行的元素的所有列都是构成超面和输入点的点。当行列式在理论上接近零时，浮点舍入误差可能引起不正确的分类。

2．分而治之算法

分而治之算法的基本概念与低维情形是一样的。重要的函数是Merge，它用于计算两个其他凸包所生成的凸包。虽然很容易描述，但是实现这种算法却非常困难。必须找到支持这两个凸包的超平面。用不属于这两个包中的任一个包的一条超边来围卷这个超平面，直到遇到任一条边的一个顶点为止。这个顶点和这条超边用来构成合并包的一个超面。围卷继续进行，直到遇到原来的超边为止。必须删除对任一个输入包可见的超面。与三维问题一样，位于一个包上的这样的超面的集合是互连的，因此深度有限搜索就足以找到并删除它们。当然，应该在合并包中保留位于每一个输入包上的隐藏超面。

13.8　德洛奈三角剖分

有限点集$S\subset\mathbb{R}^2$的三角剖分是一个三角形的集合，这些三角形的顶点都是S内的点，它们的边都连接S内的点对。S内的每一个点都要求至少在一个三角形上出现。这些边只允许相交于顶点。一个可选的要求是，这些三角形的联合是S的凸包。图 13.42 显示了两个点集的三角剖分。图 13.42（a）中的三角剖分包含了可选的要求，但是图 13.42（b）中的三角剖分却没有包含可选的要求。相似的技术可用于构建顶点都是有限集合$S\subset\mathbb{R}^3$中的点的四面体。计算几何学的研究人员将这种方法也称为三角剖分，但是也有研究者将其称为四面体剖分。对于$S\subset\mathbb{R}^d$，顶点位于S内的物体被称为单形，三角形和四面体的一般化适用于更高维的情形。我们假设单形化是一个用于描述构建顶点位于S内的单形的合适的术语。

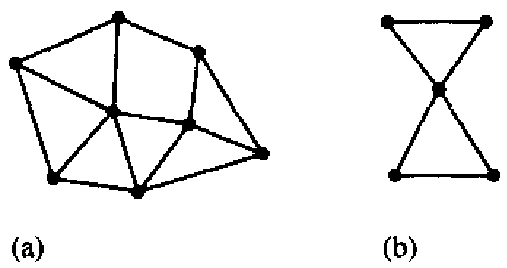


图 13.42　有限点集的三角剖分：（a）包含可选要求；（b）不包含可选要求

三角剖分的一般要求是不存在长而薄的三角形。考虑由四个点构成一个四边形的情形。图 13.43 显示了三角剖分的两种不同选择，其中顶点分别为（±2，0）和（0，±1）。其目的是选择使最小的角具有最大值的三角剖分。图 13.43（b）中的三角剖分具有这种性质。使最小的角具有最大值的方法就是德洛奈三角剖分（Delaunay triangulation）。在关于计算几何学的书籍中有一种更好的正式表述，这是在理解有限点集的 Voronoi 图的基础上提出的，并从 Voronoi 图中建立德洛奈三角剖分。一个重要的概念是外接圆，这个圆包含一个三角形的三个点。虽然可以明确地计算三角形的角，两种三角剖分法中的任一选择可由其他三个点的外接圆对一个点的包含性来等价确定。在图 13.44 中，（0，−1）位于三角形⟨(2, 0), (0, 1)，(−2, 0)⟩的外接圆内，但是（−2，0）位于三角形⟨(2, 0), (0, 1), (0, −1)⟩的外接圆外。德洛奈三角剖分具有如下性质：每一个三角形的外接圆都不包含输入集当中的其他点。这个性质用于三角剖分的增量构建中，我们将在下一节中讨论这种方法。增量式构建三角剖分的最初思想可见于 Bowyer（1981）和 Watson（1981）。

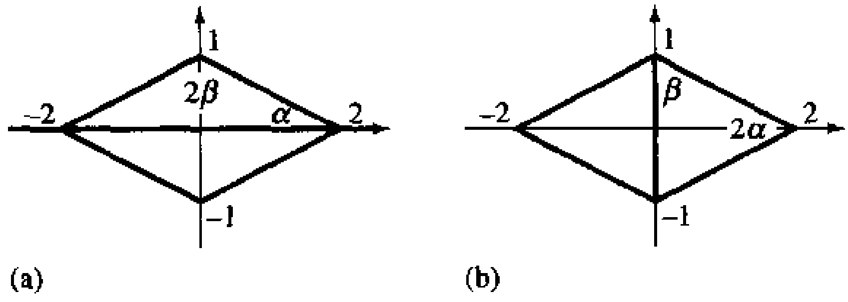


图 13.43　对一个凸四边形的两种三角剖分。角 $\alpha \doteq 0.46$ 弧度，角 $\beta \doteq 1.11$ 弧度。（a）顶部三角形的最小角是 α（小于 β）。（b）最小角是 2α 弧度（小于 β）；三角形使最小角最大化

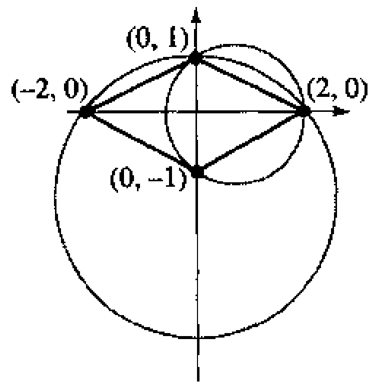


图 13.44　图 13.43 中三角形的外接圆

13.8.1　二维增量构建

给定一个$\mathbb{R}^2$上的有限点集，三角剖分开始于构建一个能包含这个点集的足够大的三角形。实际上，这个三角形应该足够大，使得它能够包含最终的三角剖分的外接圆的并集。这个三角形被称为这个点集的超级三角形。存在无数的超级三角形。例如，如果已有一个超级三角形，那么任何包含它的三角形也是一个超级三角形。容易计算的、适当大小的三角形是一个包含一个圆的三角形，而这个圆包含这些点的轴对齐有界矩形。

每一个输入点P都被插入这个三角剖分内。必须找到一个包含P的三角形。对所有三角形进行一次迭代并进行点在三角形内的检测，就能很明确地找到一个三角形，但是，当输入点集具有大量的点时，这可能是一种很慢的处理方法。更好的搜索方法是对当前的三角形进行线性行走。在 13.4.2 节中，这类方法已用于处理凸多面体的网格。当然，这种方法只适用于处理所有的多边形都是三角形的情形。如图 13.45 所示，可能出现两种情形。如果P位于三角形T之内，该三角形将被分解为三个子三角形即N_i（$0 \leqslant i \leqslant 2$）。在分解之前，这个算法的不变的部分是每一个三角形对$\langle T, A_i \rangle$，其中A_i是T的相邻三角形，都满足空间的外接圆条件。也就是说，T的外接圆不包含A的相对顶点，并且A的外接圆不包含T的相对顶点。在分解之后，T将被删除，因此三角形A_i现在与新三角形N_i相邻。可能不能满足三角形对$\langle N_i, A_i \rangle$的外接圆条件，因此需要对三角形对进行处理，以确保通过交换共用的边能够满足这个条件，图 13.43 说明了这种类型的操作。

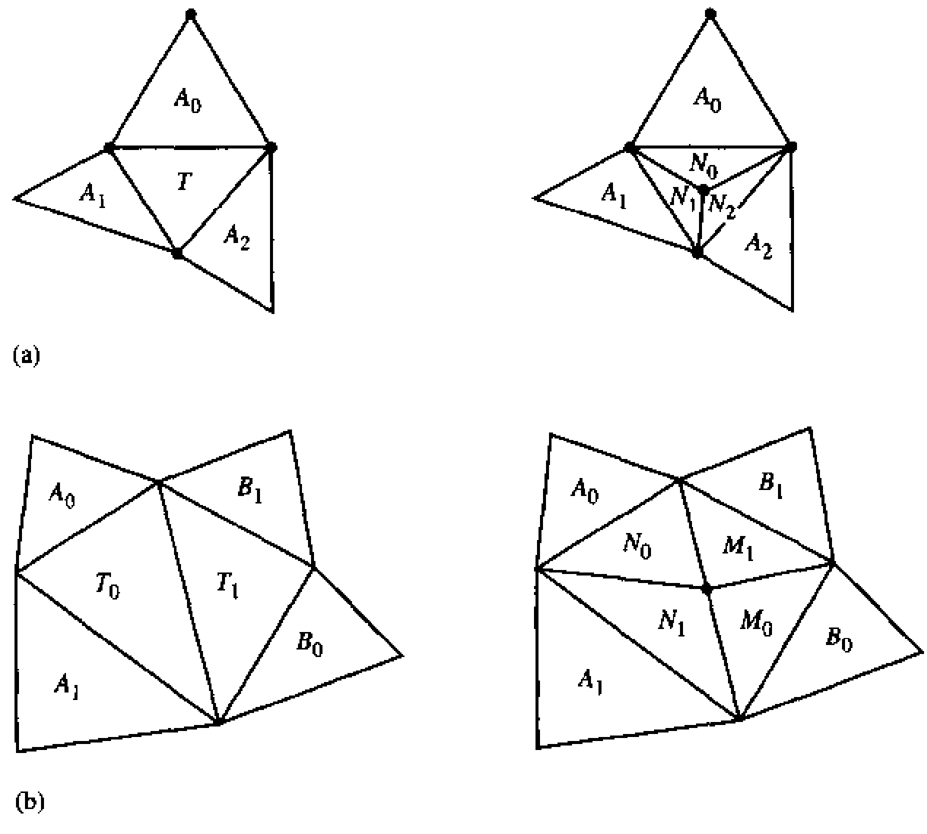


图 13.45　（a）新插入的点P（显示为一个未标记的黑点）是三角形的内部点，这种情形中三角形被分解为三个子三角形。（b）该点位于三角形的一条边上，这种情形中共用该边的每个三角形（如果存在的话）被分解为两个子三角形

增量构建的伪码列出如下：

```
void IncrementalDelaunay2D(int N, Point P[N])
{
    mesh.Insert(Supertriangle(N, P));  // mesh vertices V(0), V(1), V(2)

    for (i = 0; i < N; i++) {
        C = mesh.Container(P[i]);  // triangle or edge
        if (C is a triangle T) {
            // P[i] splits T into three subtriangles
            for (j = 0; j < 3; j++) {
                // insert subtriangle into mesh
                N = mesh.Insert(i, T.v(j), T.v(j + 1));

                // N and adjacent triangle might need edge swap
                A = T.adj(j);
                if (A is not null)
                    stack.push(N, A);
            }
            mesh.Remove(T);
        }  else {
            // C is an edge E
            // P[i] splits each triangle sharing E into two subtriangles
            for (k = 0; k <= 1; k++) {
                T = E.adj(k);
                 if (T is not null) {
                     for (j = 0; j < 3; j++) {
                         if (T.edge(i) is not equal to E) {
                             // insert subtriangle into mesh
                             N = mesh.Insert(i, T.v(j), T.v(j + 1));

                             // N and adjacent triangle might need edge swap
                             A = T.adj(j);
                             if (A is not null)
                                 stack.push(N, A);
                         }
                     }
                     mesh.Remove(T);
                 }
            }
        }

        // Relevant triangles containing P[i] have been subdivided.  Now
        // process pairs of triangles that might need an edge swapped to
        // preserve the empty circumcircle constraint.
        while (stack is not empty) {
            stack.pop(T, A);

            // see Figure 13.45 to understand these indices
            compute i0, i1, i2, i3 with T.v(i1)= A.v(i2) and T.v(i2)= A.v(i1);
```

```
                    if (T.v(i0) is in Circumcircle(A)) {
                        // swap must occur
                        N0 = mesh.Insert(T.v(i0), T.v(i1), A.v(i3));
                        B0 = A.adj(i1);
                        if (B0 is not null)
                            stack.push(N0, B0);  now <N0,B 0> might need swapping

                        N1 = mesh.Insert(T.v(i0), A.v(i3), A.v(i2));
                        B1 = A.adj(i3);
                        if (B1 is not null)
                            stack.push(N1, B1);  now <N1, B1> might need swapping
                    }
                }
            }

            // remove any triangles that share a vertex from the supertriangle
            mesh.RemoveTrianglesSharing(V(0), V(1), V(2));
        }
```

函数Supertriangle计算本节的第一个图所描述的三角形。mesh对象表示存储为顶点一边一三角形表的三角形网格。网格操作Insert接受一个三角形作为输入，或者整个地或者作为三个索引被插入网格所管理的顶点集中。它返回被插入到网格中的真正的三角形对象的一个引用，这样可以在后面的调用中使用这个对象。网格操作Remove只是删除指定的三角形。假设三角形对象T具有一个操作T.v(i)，这个操作允许访问三角形共用的网格顶点的索引。这些索引被排序，使得三角形的顶点按逆时针方向排序。索引进行与3的模运算，因此T.v(0)和T.v(3)是相同的整数。还假设三角形对象具有操作T.adj(i)，这个操作返回共用边<T.v(i),T.v(i + 1)>的相邻三角形的一个引用。如果这条边没有相邻的三角形，那么这个引用为空。假设边对象E具有操作E.adj(i)，它返回相邻的三角形的一个引用（最多有两个这样的三角形）。

图13.46显示了在处理三角形对的堆栈的循环中，伪码的不同数量。理论上，如果一个三角形对的所有顶点都位于一个共用的外接圆上，这个循环可能是无限的。下面是计算几何学所指定的典型条件：输入集中的任何4个点都不可能共圆。然而，当使用浮点算术时，必须有一种实现防止这样的情形出现。可能出现如下的情形，即4个点并不精确共圆，但是浮点舍入误差可能使三角形对表现为这些点是共圆的。如果在循环中再次遇到这样的三角形对，则它们很可能被交换回原来的构形中，并结束这个处理。为了避免无限循环，可以进行两次外接圆检测，一次针对当前的三角形构形，另一次针对交换构形。如果两次检测都说明会出现一次交换，那么就不进行交换。请注意，如果外接圆检测接受两个相邻的三角形作为输入，那么必须小心，要按相同次序来传递三角形。否则，与三角形有关的浮点计算可能导致返回相反的布尔值。也就是说，如果函数原型为bool FirstInCircumcircleOfSecond(Triangle, Triangle)，那么调用FirstInCircumcircleOfSecond(T0,T1)和FirstInCircumcircleOfSecond(T1,T0)可能返回不同的值。获得一致次序的方法如下。与三角形的共用边相对的两个顶点具有由网格对象所维护的索引。传递这个三角形，使得第一个三角形具有在两个三角形之内的最小索引的顶点。

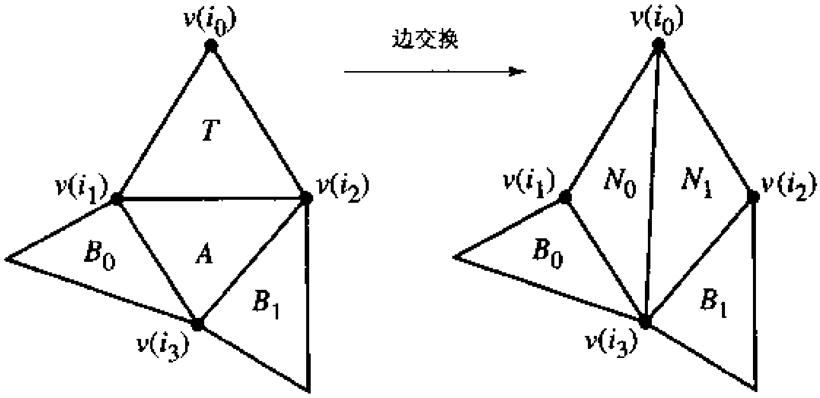


图 13.46　一个需要交换边的三角形对(T, A)。为了处理网格的顶点－边－三角形表中的正确对象，必须跟踪它们的索引。交换边后，最多将产生两对三角形，即(N_0, B_0)和(N_1, B_1)，每对三角形都可能需要交换边。这些三角形对都要压入待处理的三角形对堆栈中

一旦所有的顶点都被插入三角剖分中，并且已经交换所有需要交换的边，那么必须删除仅仅由于共用一个超级三角形的顶点而存在的那些三角形。另一种可能的缺陷是，所有的三角形都可能被删除。当初始点都位于同一条直线上时，就可能出现这种情形。如果一个应用程序确实需要用增量算法来处理这样的集合（基于数据点集的插值就是这样的一种应用），那么必须进行一些类型的预处理，以检测共线性。可能最好的方法就是利用二维德洛奈三角剖分和三维凸包之间的关系，我们将在后面描述这种关系。由于输入点集的内在维度小于点集所在空间的维数，因此凸包算法似乎更适合于处理退化情形。

13.8.2　一般维度增量构建

将二维空间的边交换算法扩展到三维空间需要考虑几个技术问题。第一个技术问题与用于寻找一个包含三角形的线性行走的扩展有关。每一个输入点P都被插入当前的四面体网格中。使用与用于二维问题相同的行走方法来寻找包含P的四面体。选择一个初始的四面体。如果P位于该四面体内，就完成行走。如果不在四面体内，就构建一条射线，它的原点是当前四面体的顶点的平均值，称之为C，它的方向为$P - C$。定位与射线相交的一个四面体的面。与当前四面体相邻并共用这个面的四面体就是包含P的下一个候选四面体。在实际中，存在病态的数据集，使得行走变得非常长（Shewchuk 2000）。一个三维的德洛奈三角剖分可能具有$O(n^2)$个多面体。在引用的论文中，已经证明了一条直线能影响$O(n^2)$个多面体。实际上，对于d维空间，一条直线能影响$n^{\lceil d/2 \rceil}$个单形的次序。

另一个技术问题是，在三维空间中并不存在交换机制的对应方法。二维空间的边交换是直观地基于使最小的角最大化来实现的，但是在三维空间中并不存在这种启发式机制（Joe 1991）。通用的方法以 Watson 算法（Watson 1981）为基础。这种算法是针对包含于内在维度为$\mathbb{R}^d$的一个点集来描述的。不幸的是，必须有一种实现处理维数的退化。实现三维空间的 Watson 算法时，Field（1986）提出了对于实际问题的一种非常好的描述。这篇论文可以通过 ACM Digital Library 在线获得。

由于已对完整的维度做了假设，因此三角剖分的元素都是没有退化的单形，每一个都具有$d+1$个点。单形的外接球面叫做外接超球。德洛奈三角剖分的条件就是一个单形的外

接超球不包含这个单形的顶点之外的任何输入点（这是外接球面为空的限制条件）。下面分几步来描述这个算法。根据 Field（1986）的精彩说明，这些步骤提供了说明在二维空间出现时的相关图形。

（1）构建一个超级单形，即能保证包含输入点和最终的三角剖分的外接超球的单形。一种适当的选择是，包含本身就包含输入点的轴对齐有界箱的超级球的单形（如图 13.47 所示）。

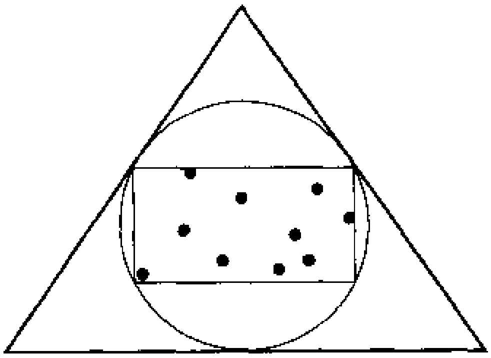


图 13.47　输入点集的超级三角形

（2）将超级单形作为第一个单形加入到单形网格中。该网格包含顶点和单形之间所有必需的互连性。而且，保存每一个单形的外接超球，以避免在增量更新时重复计算中心和半径。

（3）插入其他的输入点，一次插入一个，并进行如下的处理。

- 确定哪一个外接超球包含给定的点。下面是满足空的外接超球条件的直接要求。在插入之前，对所有单形都满足空条件。当点被插入时，必须修改对应于违背空条件的外接超球的单形。对包含超球的搜索可以按如下的方法来实现：使用上一节描述的（试图进行的线性）行走来搜索单形。但是正如 Shewchuk（2000）所指出的，最坏情形的渐近次序的影响是 $O(n^{\lceil d/2 \rceil})$。一旦找到包含单形（或者输入点位于共享的边界对象上时的单形），那么可以进行深度优先搜索以找到其他的外接超球包含输入点的单形。仅对所有当前的单形进行迭代的更简单的方法是很容易编程实现的，但是对于实际中使用的数据集来说，这种方法可能更慢（如图 13.48 所示）。
- 外接超球包含输入点的单形的并集构成一个 d 维的多面体，这个多面体叫做插入多面体。定位这种多面体的边界面（如图 13.49 所示）。
- 通过连接输入点与边界面来创建新的单形。然后删除并集是插入多边形的老的单形（如图 13.50 所示）。

（4）在所有点都被插入之后，将从网格中删除共用超级单形的一个顶点的单形。得到的网格就是这些点的德洛奈三角剖分（如图 13.51 所示）。

三维德洛奈三角剖分可能产生长条（slivers），即体积几乎为零的四面体（针状或者平的）。近年来，研究人员试图发展一些修改最终的三角剖分并获得好的质量的方法（Dey, Bajaj 和 Sugihara 1991；Cheng 等人 2000）。

如果输入点集的内在维度小于点所在的空间的维数，一种可能的更好方法是利用下一节描述的 d 维空间中的德洛奈三角剖分与 $d+1$ 维空间中的凸包之间的关系。凸包算法似乎

比三角剖分算法更适合处理维数退化的情形。

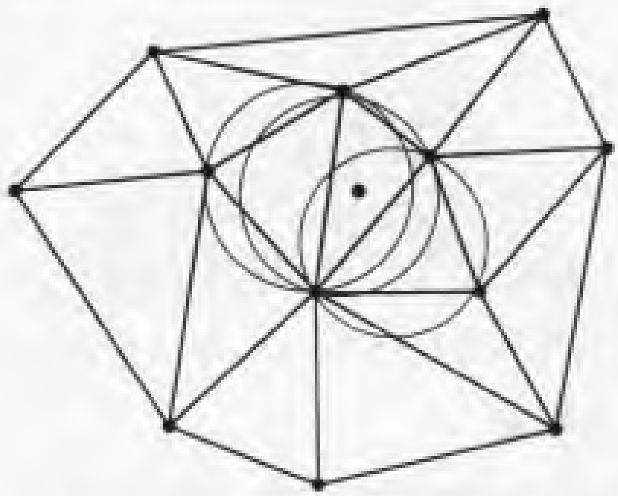


图 13.48　包含将被插入的下一点的外接圆

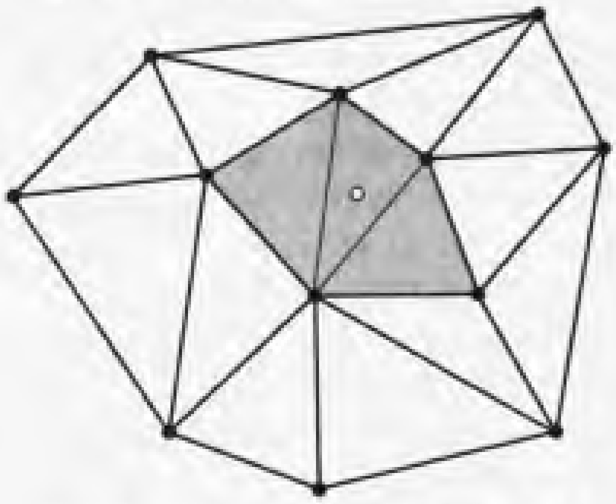


图 13.49　针对将被插入的下一点的插入多边形

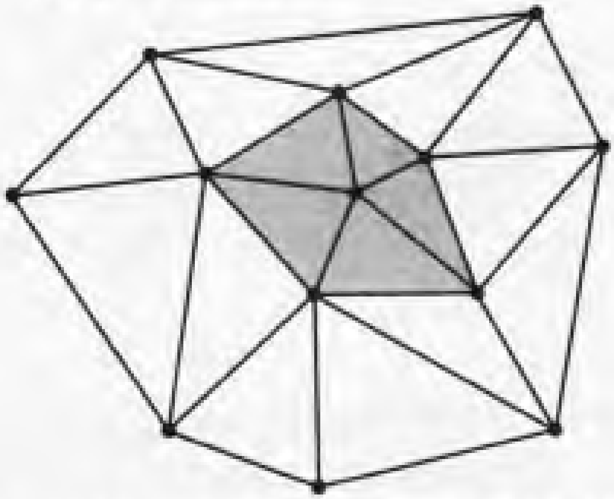


图 13.50　对整个网格恢复了空的外接圆条件而修正的插入多边形

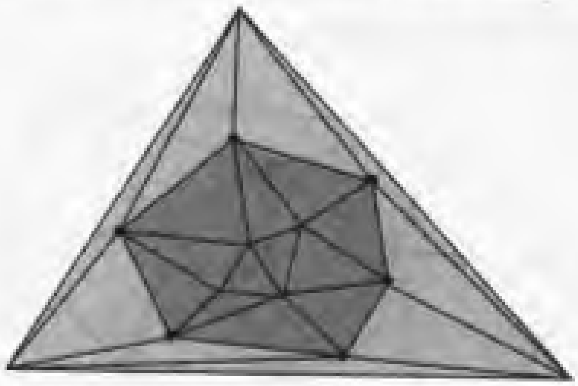


图 13.51　最终的网格三角形显示为深灰色。删除了的三角形显示为浅灰色

13.8.3　用凸包实现构建

任何维度d的有限点集$S \subset \mathbb{R}^d$的德洛奈三角分解都可以从$S' = \{(X, \|X\|^2) : X \in S\} \subset \mathbb{R}^d \times \mathbb{R} = \mathbb{R}^{d+1}$的凸包中获得，如 Edelsbrunner 和 Seidel（1986）所述。特别地，假设凸包被构建，并且其超面为$(d+1)$维的单形。每一个单形都具有$\mathbb{R}^d \times \mathbb{R}$内的法线向量，即$(\vec{N}, \lambda)$。$\lambda < 0$的单形构成所谓的低包。其他的单形构成高包。低包的单形在$\mathbb{R}^d$上的投影本身就是单形（维数为d），并且是S的德洛奈三角剖分。

图 13.52 显示了一个从三维凸包中获得的二维三角剖分的简单说明。其 5 个输入点分别是（0，0），（0，±1）和（1，±1）。图 13.52（b）显示了德洛奈三角剖分。图 13.52（a）显示了提升点（0，0，0），（0，±1，1）和（1，±1，2）。低包由三个三角形所构成。逆时针次序的三角形$\langle(0,0,0),(0,1,1),(1,1,2)\rangle$具有法线向量（1，1，−1）。第三个分量是负的，因此位于低包上，并被投影到xy平面上，以获得逆时针次序的德洛奈三角形$\langle(0,0),(1,1),(0,1)\rangle$。类似地，三角形$\langle(0,0,0),(1,-1,2),(0,-1,1)\rangle$和$\langle(0,0,0),(1,1,2),(1,-1,2)\rangle$具有第三个分量为负的法线，因此它们的$xy$平面投影是三角剖分的一部分。逆时针次序的三角形$\langle(0,1,1),(0,0,0),(0,-1,1)\rangle$具有法线向量（−2，0，0）。第三个分量为零，因此它不是低包的一部分。投影是一个退化的三角形，并且不影响三角剖分。高包由两个被抛弃的三角形所构成。

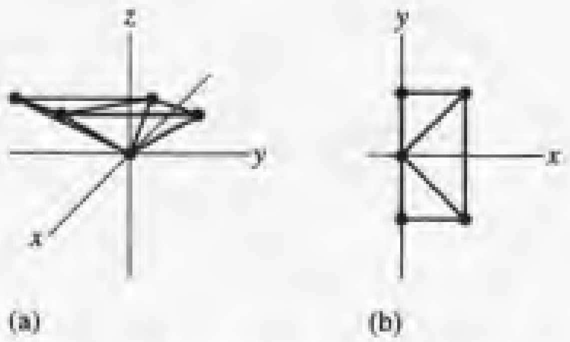


图 13.52　（a）二维点的凸包提升为三维的抛物面。（b）对应的德洛奈三角剖分，较低的包在xy平面上的投影

从数值的观点来看，这一结果特别有用。根据经验，在具有浮点数的系统中实现一个完全健壮的凸包算法，比实现一个完全健壮的德洛奈三角剖分算法要简单。

13.9　多边形分解

本节描述几个有用的将简单多边形分解为三角形或者凸多边形的算法。其中的一个关键的概念是其他的顶点对于给定顶点的可见性。两个顶点V_i和V_j被称为是互相可见的，如果连接它们的开放线段严格地位于多边形内。也就是说，从一个顶点到另一个顶点之间必须有一个清晰的视线，没有任何的多边形部分会阻挡视线，甚至没有一个顶点会阻挡视线。当两个顶点是互相可见的时，连接它们的线段叫做多边形的对角线（diagonal）。图 13.53 说明了两个顶点之间的可见性。根据定义，多边形的边并不是对角线。然而，在某些应用中，允许将边标记为对角线是非常有用的。在讨论中，我们区分顶点的类型。如果两条共用一个顶点的边之间的夹角（从多边形内测量）小于弧度π，那么这个顶点就称为凸顶点。这个角称为位于顶点上的内角。如果共用一个顶点的两条边位于同一条直线上，这个顶点就叫做共线顶点。在典型的应用中，这些顶点不能用于构建多边形，或者在进行某些修改多边形的操作之后被删除。如果位于一个顶点上的内角大于弧度π，那么这个顶点称为优角。

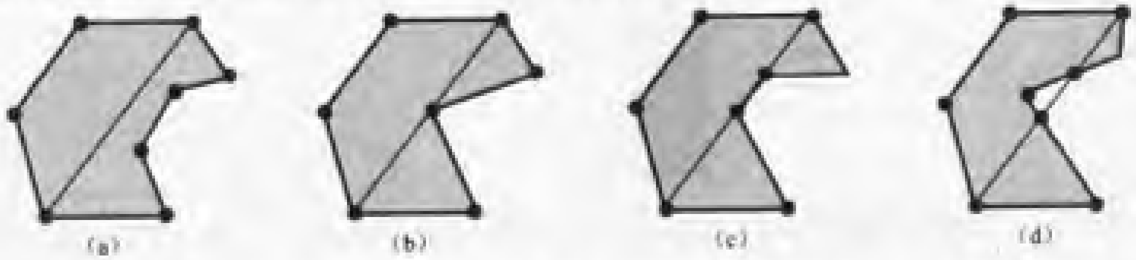


图 13.53　(a) 互相可见的两个顶点，图中画出了连接它们的对角线。(b) 互相不可见的两个顶点，被它们之间的一个顶点阻挡。(c) 互相不可见的两个顶点，被它们之间的一条边阻挡。(d) 互相不可见的两个顶点，被多边形外的一个区域阻挡

13.9.1　一个简单多边形的可见性图

给定一个顶点次序为从V_0到V_{n-1}的简单多边形，可以构建一个无向图，其节点表示这个多边形的顶点，其弧表示该多边形的对角线。这个图叫做该多边形的可见性图。在这个图中，不允许存在从一个顶点连接到它自己的弧。如果这个图表示为一个邻接矩阵，如果V_i和V_j是互相可见的，则项(i, j)为 1，否则就为 0。当然，这个矩阵是对称的，因为图是无向的。一个凸多边形的这种图由三个零对角线组成，即主对角线、主对角线的子对角线和主对角线的超对角线，但是所有其他的项都为一。对于其他的简单多边形，该矩阵可能很复杂。在多边形分解中，可能需要对可见性图的部分或者全部进行比较。

1. V_{i-1}和V_{i+1}之间的可见性

对于单形，假设其顶点索引通过与n的模运算来计算，n为多边形顶点的数量。我们从可见性确定的最简单情形开始。给定一个顶点V_{i-1}，我们希望确定V_{i+1}对该点是否是可见的。

如果V_i是一个优角顶点，那么连接V_{i-1}和V_{i+1}的线段至少必须是部分位于多边形之外的，因此V_{i-1}和V_{i+1}并不是互相可见的。一般认为仅有V_i是凸角顶点就足够了。对于这样

的顶点，只要没有多边形上的边与其相交，那么线段(V_{i-1}, V_{i+1})就是对角线。几何意义上与此等价的是，不存在其他的多边形顶点位于三角形(V_{i-1}, V_i, V_{i+1})上。如果多边形具有r个优角顶点和$n-r$个凸角顶点，那么直白的实现将处理$n-r$个三角形中的每一个，并最多进行$n-3$次点在三角形内的检测。如果(V_{i-1}, V_{i+1})确实是一条对角线，就会出现最大的检测次数。这种算法的时间级为$O((n-r)n)=O(n^2)$。一种更有效的实现可以避免检测三角形对所有多边形顶点的包含性。如果这个多边形的边界与线段(V_{i-1}, V_{i+1})相交，那么边界的位于三角形内的部分必须包含至少一个优角顶点。这就说明，多边形边界必须进入三角形，稍后再拐弯退出三角形。这个拐弯要求形成一个优角顶点。因此，仅仅检测三角形对这个优角顶点的包含性就足够了。这个算法的时间级为$O(nr)$，当r比n要小很多时，这要比$O(n^2)$好得多。图 13.54 说明了这种思想。

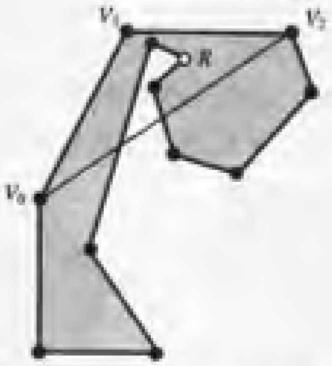


图 13.54　图示说明为什么V_0与V_2之间缺少可见性等价于三角形(V_0, V_1, V_2)包含一个优角顶点R

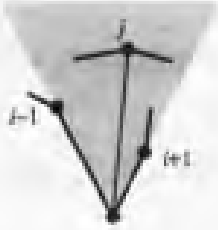
如果(V_{i-1}, V_{i+2})是多边形的一条对角线，顶点V_{i-1}, V_i和V_{i+1}就将构成一个多边形的耳，而V_i叫做耳尖（ear tip）。稍后讨论的一种多边形的三角剖分方法利用了对耳的搜索方法。在图 13.54 中，顶点V_0就是一个耳尖，而这个耳就是由V_0和它的两个相邻顶点所构成的三角形。

2. 任意两个顶点之间的可见性

检测(V_i, V_j)（$|i-j|\geqslant 2$）是不是对角线的方法非常简单。这种方法的基本思想是，遍历多边形的所有边，并检测是否存在任何与线段(V_i, V_j)相交的边。不需要检测与指定的线段相邻的边。最差的情形是相邻的边与线段共线，在这种情形中，当前的相邻边的下一条相邻边有一个端点在这条线段上。进行与这条边的相减测试就能找到交点。

如果多边形有一条边与指定的线段相交，那么线段就不是一条对角线。然而，如果所有的边（与线段相邻的边除外）与线段都不相交，那么存在两种情形。一种情形是这条线段是一条对角线。另一种情形是这条线段位于多边形之外，进行对角线检测的代码必须区分这两种情形。确定线段的一个端点（比如V_i）的局部表现就足够了。只要线段包含于顶点为V_i且边方向为$V_{i\pm 1}-V_i$的圆锥之内，这条线段就是一条对角线。图 13.55 显示了凸角顶点和优角顶点的包含条件。

检测(V_{i_0}, V_{i_1})是否是对角线的伪码列出如下。最前面的两个输入都是简单多边形。最后的两个输入都是将要检测的线段的端点的索引。假设是在$|i_0-i_1|\geqslant 2$中对索引进行与n的模运算。



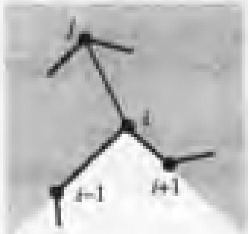


图 13.55　(a) 凸角顶点的圆锥包容；(b) 优角顶点的圆锥包容

```
bool IsDiagonal(int n, Point V[n], int i0, int i1)
{
    // Segment may be a diagonal or may be external to the polygon.  Need
    // to distinguish between the two.  The first two arguments of
    // SegmentInCone are the line segment.  The first and last two arguments
    // form the cone.

    iM = (i0 - 1) mod n;
    iP = (i0 + 1) mod n;
    if (not SegmentInCone(V[i0], V[i1], V[iM], V[iP]))
        return false;

    // test segment <V[i0], V[i1]> to see if it is a diagonal
    for (j0 = 0, j1 = n - 1; j0 < n; j1 = j0, j0++) {
        if (j0 != i0 && j0 != i1 && j1 != i0 && j1 != i1) {
            // The first two arguments of SegmentsIntersect form a line
            // segment.  The last two arguments form an edge to be tested
            // for intersection with the segment.

            if (SegmentsIntersect(V[i0], V[i1], V[j0], V[j1]))
                return false;
        }
    }
    return true;
}
```

如果交点存在，那么可以通过计算任何一个交点来实现函数 SegmentIntersect。否则，可以通过计算与这条线段之间的距离来实现函数 SegmentIntersect。6.6 节和 7.1 节讨论了这些话题。函数 SegmentInCone 的伪码显示如下。

```
float Kross(Point U, Point V)
{
    // Kross(U, V) = Cross((U,0), (V,0)).z
    return U.x * V.y - U.y * V.x;
}

bool SegmentInCone(Point V0, Point V1, Point VM, Point VP)
{
    // assert:  VM, V0, VP are not collinear

    Point diff = V1 - V0, edgeL = VM - V0, edgeR = VP - V0;
```

```
        if (Kross(edgeR, edgeL) > 0)  {
            // vertex is convex
            return (Kross(diff, edgeR) > 0 and Kross(diff, edgeL) < 0);
        } else {
            // vertex is reflex
            return (Kross(diff, edgeR) < 0 or Kross(diff, edgeL) > 0);
        }
    }
```

Meister（1975）提出了关于对角线和耳的两个事实。第一个事实是，一个至少有 4 个顶点的多边形具有至少一条对角线。第二个事实被称为 Meister 的二耳定理，即一个至少有 4 个顶点的多边形必定至少具有两个非重叠的耳。

13.9.2　三角剖分

设有一个具有顶点V_i（$0 \leqslant i < n$）的简单多边形。这个多边形的三角剖分（triangulation）就是对这个多边形进行三角形分解。每一个三角形的顶点都是原多边形的顶点。如果分解形成的两个三角形相交，那么它们只能相交于顶点或边，而不能相交于内点。这种处理所生成的三角形的边必须位于多边形之内，因此这些边必须是对角线。根据定义，所有用于三角剖分的对角线都必须是不相交的。这就说明，多边形的任何三角剖分都必须使用$n-3$条对角线，并包含$n-2$个三角形。

可能出现较少数量的三角形的情形，这些三角形的并集也是原来的多边形，但是，在这种情形中将出现 T 形连接。这就等价于必须使用非对角线的线段来连接顶点。例如，具有有序顶点$V_0=(0,0)$，$V_1=(1,-1)$，$V_2=(1,1)$，$V_3=(-1,1)$，$V_4=(-1,-1)$的多边形可以分解为三个三角形，它们的索引三元组分别为$\{0,1,2\}$，$\{0,2,3\}$和$\{0,3,4\}$。用于三角剖分的对角线为$\{0,2\}$和$\{0,3\}$。只有两个三角形的分解为$\{0,1,2\}$和$\{2,3,4\}$，但是V_0是 T 形连接，而且线段$\{2,4\}$不是对角线。在应用中一般不希望出现 T 形连接。

通过耳剪裁来实现三角剖分

因为多边形的三角剖分与多边形的对角线相关，因此可以使用分而治之算法来构建三角剖分。其思想是，找到一条对角线，沿对角线将多边形分解为两个子多边形，并对每一个子多边形进行递归处理。伪码列出如下，其中的顶层调用传入一个多边形，这个多边形存储为一个互连的列表和一个空的索引三元组列表。假设这个多边形至少具有三个顶点。

```
void Triangulate(VertexList vlist, TriangleList tlist)
{
    n = vlist.size;
    if (n == 3) {
        tlist.Add(vlist(0), vlist(1), vlist(2));
        return;
    }

    for (i0 = 0; i0 < n; i0++) {
        for (i1 = 0; i1 < n; i1++) {
            if (IsDiagonal(vlist, i0, i1))  {
                Split(vlist,sublist0, sublist1);
```

```
                Triangulate(sublist0, tlist);
                Triangulate(sublist1, tlist);
                return;
            }
        }
    }
}
```

其中的双重循环是$O(n^2)$级的，而对角线检测是$O(n)$级的，因此用这种方法寻找对角线是$O(n^3)$级的。利用互连的列表，分解操作是$O(1)$级的。因此，在对子多边形进行三角剖分之前需要花费$O(n^3)$级的时间。粗略分析，如果每一个子多边形具有$n/2$个顶点，那么对于需要时间T_n的递归公式，求解源问题所需的时间为$T_n = 2T_{n/2} + O(n^3)$。应用 Cormen，Leiserson 和 Rivest（1990）中的主定理，可得$T_n = O(n^3)$。

使用耳剪裁的一种变体可能更简单一些。不需要搜索任何的对角线，只需要找到一个耳就足够了，将对应的三角形加入列表中，从多边形中删除这个耳，并对减少的多边形进行递归处理。伪码如下：

```
void Triangulate(VertexList vlist, TriangleList tlist)
{
    n = vlist.size;
    if (n == 3) {
        tlist.Add(vlist(0), vlist(1), vlist(2));
        return;
    }

    for (i0 = 0, i1 = 1, i2 = 2, i0 < n;
         i0++, i1 = (i1 + 1) mod n, i2 = (i2 + 1) mod n) {
        if (IsDiagonal(vlist, i0, i2))   {
            RemoveVertex(vlist, i1, sublist);
            Triangulate(sublist, tlist);
            return;
        }
    }
}
```

在这种情形中，外层循环和对角线搜索的组合需要$O(n^2)$级的时间。顶点删除是$O(1)$级的。子多边形具有$n-1$个顶点，因此递归公式的处理时间为$T_n = T_{n-1} + O(n^2)$级的。这种递归所需的时间为$T_n = O(n^3)$，与前一种方法具有相同的数量级。

可以进一步修改耳剪裁算法，使其成为$O(n^2)$级的算法。其思想是，首先扫描一次多边形，并跟踪记录哪些顶点是耳尖，哪些顶点不是耳尖。需要处理的顶点数量为n，而每一个耳检测所需时间为$O(n)$级的，它们的组合需要$O(n^2)$级的时间。第二次扫描删除一个耳，比如说，删除耳尖位于顶点i的耳。这样，顶点包含顶点$i-1$和$i+1$的耳将因为这种删除而发生变换。与其他顶点相关的耳并不需改变。因此，每一次耳删除仅需要两次更新，以确定顶点$i-1$和$i+1$是否是减小的多边形的耳尖。每一次更新都与对角线检测相关，这是一种$O(n)$级的操作。第二次扫描实际上就是对$O(n)$个耳的一次递归，每一次更新需要$O(n)$级的时间用于对角线检测，因此组合后的扫描也是$O(n^2)$级的。伪码为

```
void Triangulate(VertexList vlist, TriangleList tlist)
{
    // dynamic list for polygon, to be reduced as ears are clipped
    VertexList vdynalist = vlist.Copy();  // copy entire list
    int vdynaquantity = vlist.Size();
// dynamic list for ear tips, reduced/updated as ears are clipped
VertexList node = vdynalist;
VertexList edynalist = empty;
for (i = 0; i < vdynaquantity; i++, node = node.Next())  {
    if (node is an ear of vlist) {
        VertexList tmp = node.CopySingle();  // copy only node (not links)
        if (edynalist is not empty)
            edynalist.InsertBefore(tmp);  // tmp inserted before edynalist
        else
            edynalist = tmp;  // first element in list
    }
}

// remove ears one at a time
while (true) {
    // add triangle to output list (three integer indices)
    tlist.Add(vdynalist.Previous().VertexIndex());
    tlist.Add(vdynalist.VertexIndex());
    tlist.Add(vdynalist.Next().VertexIndex());
    if (vdynaquantity == 3)
        return;  // last triangle was added, done triangulating
// remove the ear tip from vertex list
VertexList vprev = vdynalist.Previous();
VertexList vnext = vdynalist.Next();
vdynaquantity--;
vdynalist.RemoveSelf();

// Previous node to ear had a topological change.  Recompute its
// earness.
if (vprev is an ear of vlist)  {
    if (vprev.VertexIndex() != edynalist.Previous().VertexIndex()) {
        // removal of old ear caused vprev to be an ear
        edynalist.InsertBefore(vprev.CopySingle());
    }
} else {
    if (vprev.VertexIndex() == edynalist.Previous().VertexIndex()) {
        // removal of old ear caused vprev not to be an ear
        edynalist.Previous().RemoveSelf();
    }
}

// Next node to ear had a topological change.  Recompute its earness.
// Advance to next vertex/ear.
        if (vnext is an ear of vlist) {
            if (vnext.VertexIndex() != edynalist.Next().VertexIndex()) {
```

```
                // removal of old ear caused vnext to be an ear
                edynalist.InsertAfter(vnext.CopySingle());
            }
        } else {
            if (vnext.VertexIndex() == edynalist.Next().VertexIndex()) {
                // removal of old ear caused vnext not to be an ear
                edynalist.Next().RemoveSelf();
                vnext = vnext.Next();
            }
        }

        // get next vertex
        vdynalist = vnext;

        // get next ear and remove the old ear from list
        edynalist = edynalist.Next();
        edynalist.Previous().RemoveSelf();
    }
}
```

实现更好的多边形三角剖分方法是可能的。20 世纪 80 年代晚期和 20 世纪 90 年代早期的许多研究专注于这一论题。Chazelle（1991）证明了三角剖分可以在$O(n)$级时间内完成。然而，这种算法非常复杂，而且还不清楚是否可以在实践中实现这种算法。而表现较差的算法一般比较容易实现。

13.9.3 水平分解三角剖分

本节描述了一种$O(n\log n)$级的方法，这种方法的基础是将多边形分解为两条平行边平行于x轴的梯形（Chazelle 1991；Fournier 和 Montuno 1984）。对于少量的单调多边形，这种要求$O(n\log n)$级时间的分解可以进一步减少为$O(n)$级。每个单调多边形都可在$O(n)$级时间内完成。Clarkson，Tarjan，Van Wyk（1989），以及 Seidel（1991）中提出了一个更快的算法，这种算法使用随机数，并且几乎是线性的。这种算法的时间级是$O(n\log^* n)$，其中$\log^* n$用迭代对数定义如下

$$\log^{(i)}(n)=\begin{cases} n, & i=0\\ \log(\log^{(i-1)}n), & i>0 \text{ 且 } \log^{(i-1)}n>0\\ \text{undefined}, & i>0 \text{ 且 } \log^{(i-1)}n\leqslant 0 \text{ 或 } \log^{(i-1)}n \text{ 未定义}\end{cases}$$

$\log^{(i)}n$表示一个迭代函数值，并不是对数的i次幂。最后的定义为

$$\log^* n=\min\{i\geqslant 0:\log^{(i)}n\leqslant 1\}$$

这是一个增长非常缓慢的函数。例如，$\log^*(2)=1$, $\log^*(4)=2$, $\log^*(16)=3$, $\log^*(65536)=4$ 和$\log^*(2^{65536})=5$。在实际应用中，实际上$\log^* n$是一个非常小的常数。上面提到的随机算法实际上是$O(n)$级的。Paeth（1995）中的论文“基于 Seidel 算法的快速多边形三角剖分”提供了这种算法的一个简要描述。图 13.56 中显示的多边形例子用于说明这种思想，这个例子摘自上述的论文。

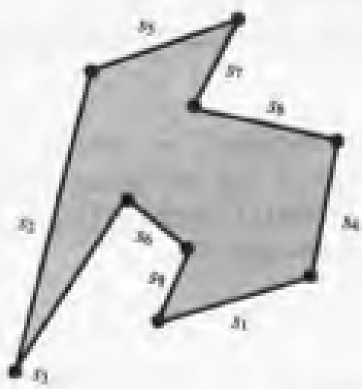


图 13.56　用于说明水平梯形分解的简单多边形。边是随机标记的，并按数字的大小次序进行处理

1．水平分解

其思想是，通过对多边形顶点的 y 值进行排序，构建一个水平条的集合。可以利用基于数组的 $O(n\log n)$ 级的排序方法来对 y 值进行直接排序，但是，对边进行随机处理，并利用二叉搜索树之类的动态数据结构来一次处理一条边，可以得到更好的效果。在一个条中，所有的子多边形都被分解为梯形，每一个条管理一个二叉搜索树之类的动态数据结构，以允许对梯形进行增量排序。将多边形看成是整个平面的分解是很方便的。这将使条和梯形列表的边界条件处理在实现中变得更简单。

我们将利用图 13.56 中的多边形作为例子来描述这个算法。假设多边形顶点的次序是逆时针方向的。我们将在这个例子之后给出相应的伪码。在这个例子中，整个 xy 平面被表示为一个矩形。开始时，第一个条和第一个梯形 T_0 就是整个平面，如图 13.57 所示。在下面的每一幅图中，分解的几何视图都位于左边，而梯形的对应图都位于右边。



图 13.57　整个平面是一个梯形

边 s_k 具有端点 (x_0, y_0) 和 (x_1, y_1)。其分量由 $s_k.x_0$，$s_k.y_0$，$s_k.x_1$ 和 $s_k.y_1$ 指定。要处理的第一条边是 s_1。其 y 值 $s_1.y$ 要求当前的条被分解为两个条，每一个条都包含一个表示一个半平面的梯形（如图 13.58 所示）。值 $s_1.y_1$ 导致另一个条被分解，每一个条都包含一个很大的梯形（如图 13.59 所示）。边 s_1 被插入，也就是说被插入到数据结构中。这种插入导致中间的条的一个梯形被分解为两个梯形（如图 13.60 所示）。

下一步要处理的是边 s_2。值 $s_2.y_1$ 将引起一次分解（如图 13.61 所示）。值 $s_2.y_0$ 将引起另一次分解（如图 13.62 所示）。s_2 的插入引起这条边所跨越的三个条中的梯形的分解（如图 13.63 所示）。

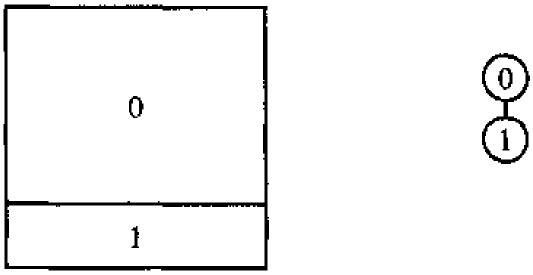


图 13.58　被 s_1-y_0 分解

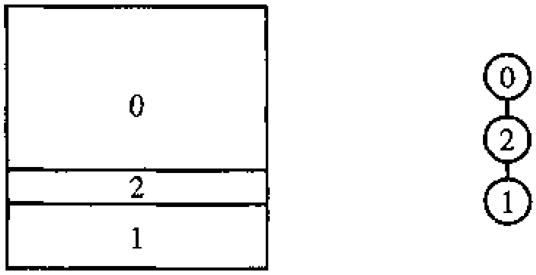


图 13.59　被 s_1-y_1 分解

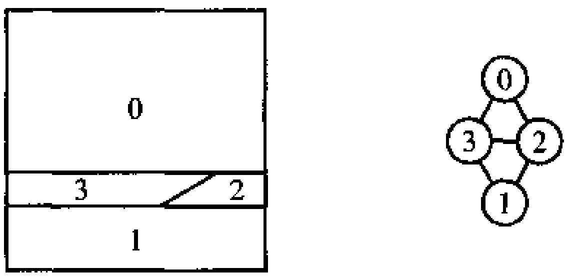


图 13.60　插入 s_1

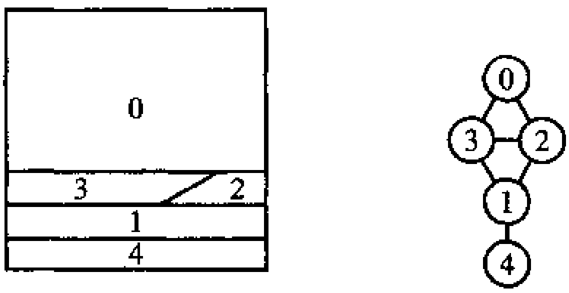


图 13.61　被 s_2-y_1 分解

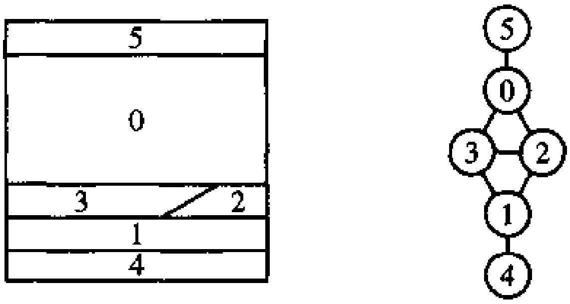


图 13.62　被 s_2-y_0 分解

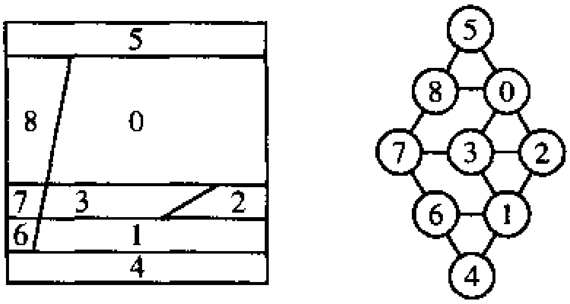


图 13.63　插入s_2

下一步要处理的是边s_3。值$s_3.y_1$将引起一次分解（如图 13.64 所示）。值$s_3.y_0 = s_2.y_1$已被处理过，因此不再出现分解。s_3的插入引起这条边所跨越的条中的梯形的分解（如图 13.65 所示）。

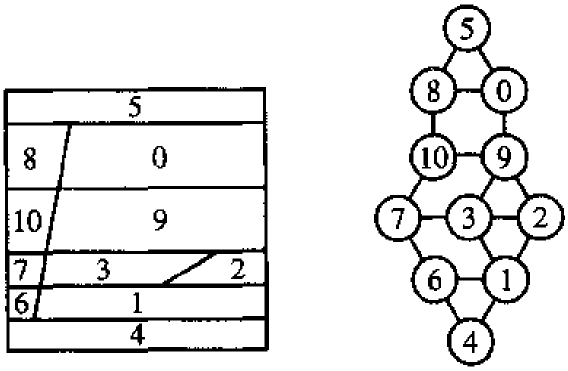


图 13.64　被$s_3.y_1$分解

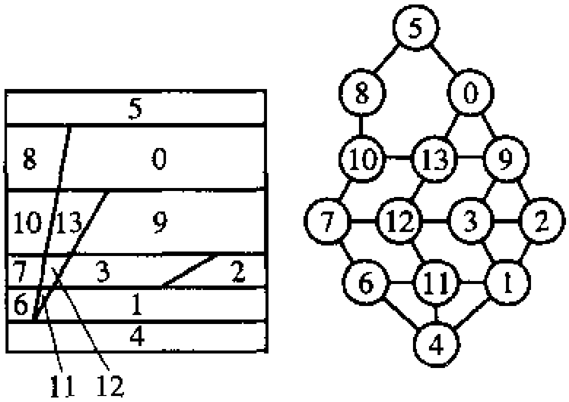


图 13.65　插入s_3

可以用相似的方法来处理剩下的边。边s_9是最后要处理的边。它的插入引起这条边所跨越的条中的梯形的分解（如图 13.66 所示）。注意到，平面已被分解为有限数量的条，每一个条都由有限数量的梯形组成。这种数据结构使用二叉搜索树，条和条内的梯形都使用二叉搜索树。这种数据结构可用于$O(\log n)$级的点在多边形内的检测。给定一个检测点，其y值用于条的二叉搜索树，以寻找包含这个y值的条。如果这个y值位于一个条的边界上，那么可以使用共用这条边界的任何一个条。梯形的二叉搜索树存储梯形左面的边和右

面的边的直线方程。将检测点的x值与这些直线进行比较，以确定这个点位于哪一个梯形上。如果这个输入的检测点位于一个条的边界上，那么只要这个梯形是一个“外面的”梯形，这种比较就应该仅限于不等比较。由于多边形的边用于分解这些条，因此说明里面／外面的标记是很容易确定的。

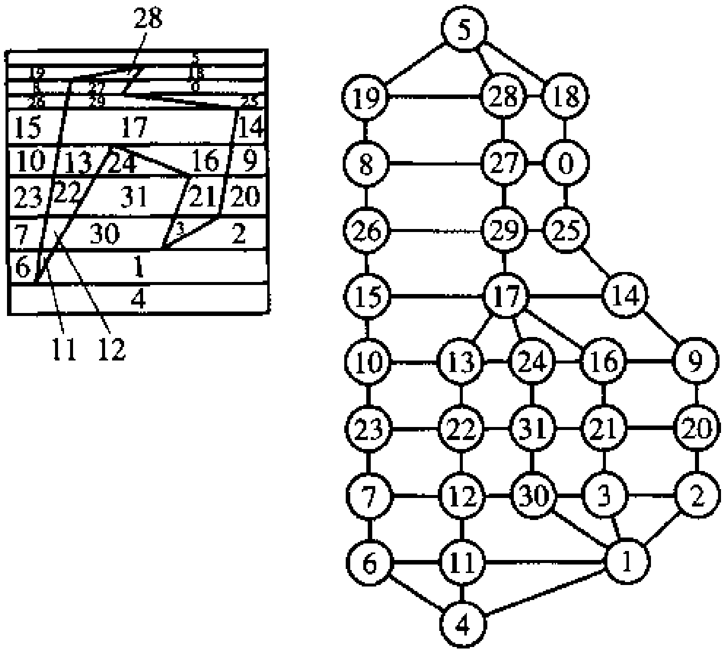


图 13.66　插入 s_9

下面列出了构建条和梯形的顶层调用的伪码。Strip 对象支持二叉搜索树数据结构，并且还有三个成员，即条的 y值的最小值min，条的 y值的最大值 max，以及Trapezoid 二叉搜索树对象 ttree。Trapezoid 对象也支持二叉搜索树数据结构，并且还有三个成员，即对应于左面的边的多边形边的索引min，对应于右面的边的多边形边的索引 max，以及方向标志 classify，如果位于最大边上的多边形的内点位于正的x方向上，则其值为+1，或者如果位于最小的边上的多边形的内点位于负的x方向上，则其值为−1。

```
void Decompose(int N, Point P[N])
{
    // randomly permute edges
    int index[N] = PermuteRange(0, N - 1);
    // prototypes: Strip(min, max, ttree), Trapezoid(min, max, classify)
    S = new Strip(-infinity, +infinity, new Trapezoid(-infinity, +infinity, -1));
    for (i0 = 0; i0 < N; i0++) {
        i1 = (i0 + 1) mod N;
        if (P[i0].y < P[i1].y)
            Insert(S,P[i0].y, P[i1].y, i, -1);  // interior in negative x-direction
        else if ( [i1].y < P[i0].y)
            Insert(S,P[i1].y, P[i0].y, i, +1);  // interior in positive x-direction

        // else ignore horizontal edges
    }
}
```

条插入的伪码为

```
void Insert(Strip S, float y0, float y1, int i, int classify)
{
    // binary search for strip containing y0, assumes S0 = [min, max)
    S0 = Locate(S, y0);
    if (y0 > S0.min) {
        // y0 interior to strip, split to N = [min, y0), S0 = [y0, max)
        N = new Strip(y0, max, S0.CopyOfTTree());
        S0.min = y0;
        S0.InsertBefore(N);  // insert N before S0 in search tree
    }

    // binary search for strip containing y1, assumes strip is min <= y < max
    S1 = Locate(S, y1);
    if (y1 > S1.min) {
        // y1 interior to strip, split to S1 = [min, y1), N = [y1, max)
        N = new Strip(y1, max, S1.CopyOfTTree());
        S1.max = y1;
        S1.InsertAfter(N);  // insert N after S1 in search tree
    }

    // add a trapezoid to each strip spanned by edge
    for (L = S0; L <= S1; L++)
        Insert(L.ttree, (L.min + L.max) / 2, i, classify);
}
```

梯形插入的伪码为

```
void Insert(Trapezoid T, float mid, int i, int classify)
{
    // Locate correct place to insert new trapezoid by comparing x-values
    // along the mid line passing through the trapezoids.
    T0 = LocateMid(T, i);

    // Split T0 = {min,max} to N = {min,i} and T0 = {i,max}
    N = new Trapezoid(T0.min, i, classify);
    T0.min = i;
    T0.classify = -classify;
    T0.InsertBefore(N);  // insert N before T0 in search tree
}
```

上述的伪码支持可用于点在多边形内查询的数据结构的构建。为了支持三角剖分，垂直相邻的梯形必须合并。这就要求在trapezoid对象中增加两个列表对象，这两个列表对象用于存储在条的最小边和条的最大边上垂直相邻的梯形的连接。必须相应地修改伪码以构建这些连接。

2. 单调多边形构建

三角剖分的下一步是构建并集为原始多边形的单调多边形。这种处理的第一步是将梯形合并为具有最大面积的梯形。图 13.67 显示了该平面的合并构形。该图显示了外部被合

并的外部梯形。然而，对于三角剖分来说，仅仅要求内部梯形。图 13.68 显示了仅适用于多边形本身的合并梯形。

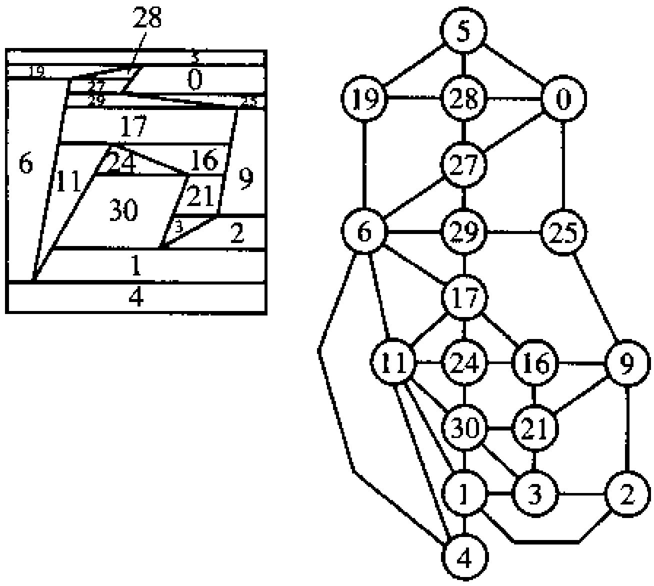


图 13.67　梯形被合并到最大的梯形内之后的平面

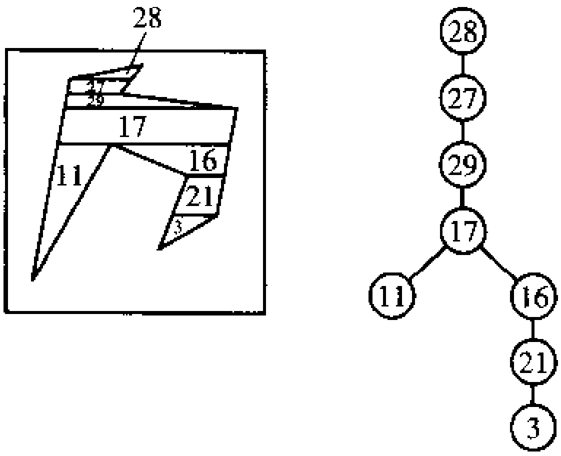


图 13.68　梯形被合并到最大的梯形内之后的多边形例了

第二步是加入连接多边形顶点的线段。这些线段和原始多边形的边形成一个分解成为单调多边形的分解，其单调性与 y 方向相关。假设原始多边形中没有任何两个顶点具有相同的 y 值。对这个多边形的水平分解所得的每一个梯形都刚好包含这个多边形的两个顶点，一个位于梯形的顶边，另一个位于梯形的底边。如果顶点是边的内点，那么它必定是一个尖点。也就是说，共用该顶点的两条边将同时位于通过顶点的水平线之上或者之下。向下开放的尖点出现在一些梯形的底边上。连接这种尖点到梯形顶边上的顶点的线段被加入。类似地，向上开放的尖点出现在一些梯形的顶边上。连接这种尖点到梯形底边上的顶点的线段被加入。新加入的线段的两个端点也可能都是尖点。图 13.69 显示了正在说明的例子。只有一条线段需要加入，即连接向下开放的尖点和位于梯形的顶边上的一个顶点的线段。这个多边形由两个单调多边形所构成。

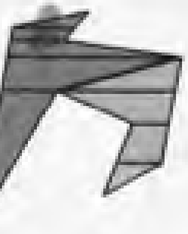


图 13.69　作为单调多边形的并的示例多边形。两个多边形分别显示为浅灰色和深灰色。梯形分解用的水平线仍然显示在图中

上述最后一段的论述是建立在一个假设的基础之上的，即不存在两个顶点具有相同的y值。当然，这种情形在实际中是可能出现的。将线段加入以形成单调多边形的算法需要做一点修改以处理这样的情形。

3．单调多边形三角剖分

在一个简单多边形的分解中获得的多边形都是与y轴相关的单调多边形。即，x方向上的直线都与y方向上的单调多边形相交于一个点（y方向上的一个极值点）、两个点（典型情形）或者沿着多边形的整条边（边是水平的）。单调多边形的这种三角剖分仅需要使用$O(n)$级的时间来计算（Fournier 和 Montuno 1984）。这种方法与一种从多边形的极值点删除三角形的费时算法有关。

首先，我们仅仅试图从多边形的一个极值点删除一个三角形，看看会出现什么问题。设V_{min}和V_{max}为多边形的极值点。这个多边形具有两条分别被称为左链和右链的单调链。设$V_0=(x_0, y_0)$为位于左链上与V_{min}相邻的顶点，并设$V_1=(x_1, y_1)$为位于右链上的相邻顶点。为了便于讨论，假设$y_0 \geq y_1$。否则，我们可以将链的地位反转并进行相同的讨论。如果三角形$\langle V_0, V_{min}, V_1\rangle$是一个耳，那么这个耳可被删除，并加入三角剖分中的三角形列表中。左链的边$\langle V_{min}, V_0\rangle$被删除，并被边$\langle V_1, V_0\rangle$所替换。修改后的链在$y$方向上仍然是单调的，因此这个过程可以在位于其最小值顶点的新的单调多边形上重复进行。图 13.70 提供了一种说明。

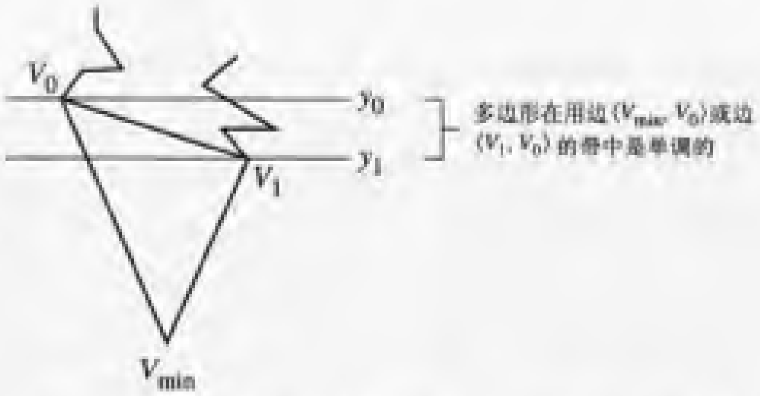


图 13.70　如果位于极值点的三角形是耳，则删除产生另一个单调多边形的耳

这种方法中的问题是它要求确定位于最小值顶点上的三角形是否是一个耳。实际上，最小值顶点可能是一个耳尖。图 13.71 显示了两种可能出现的失败类型。由耳尖而导致失

败，仅仅是因为位于右链 $\langle V_0, W\rangle$ 上的下一条边位于三角形 $\langle V_0, V_{min}, V_1\rangle$ 内（图 13.71（a））。这种失败也可能是由于更加复杂的原因，例如，位于条 $y_1 < y < y_0$ 内顶点的一条链位于刚才提到的三角形之外，但是在这条链上的下一个顶点W却位于三角形之内（图 13.71（b））。

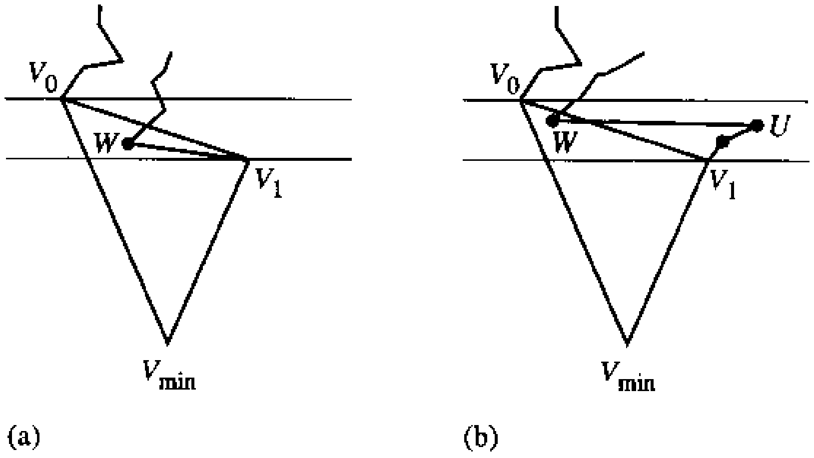


图 13.71　三角形 $\langle V_0, V_{min}, V_1\rangle$ 成为一个耳的失败情形

图 13.71（a）中的构形很容易处理。三角形 $\langle V_{min}, V_1, W\rangle$是一个耳，并且可被删除。位于右链上的边 $\langle V_{min}, V_1\rangle$ 被删除并被 $\langle V_{min}, W\rangle$ 所替换。右链仍然是单调的，因此减小后的多边形是单调的，并可以重复这种处理。即使W没有位于 $\langle V_0, V_{min}, V_1\rangle$之内，只要 V_1 是一个凸角顶点，仍然可以出现这种删除。

图 13.71（b）中的构形更加令人感兴趣。其中出现的一系列优角顶点被称为优角顶点链。在图中，W 的前一个顶点 U 是优角顶点链出现之后的第一个顶点，因此它是一个凸角顶点，紧接着的下一条具有端点 W 的边使得这条优角顶点链对 V_0 不可见。然而，位于优角链上的所有顶点对 W 都是可见的，因此由 W 和优角顶点链上的顶点构成的三角形都可被删除。右链减小为单调链，它的最前面的三个顶点为 V_{min}，W 和 U。减小后的多边形仍然是单调的。

即使如此，图 13.71（b）中的构形也并不是其他构形的代表。顶点W可能不位于三角形 $\langle V_0, V_{min}, V_1\rangle$上，但是由它和优角顶点链上的顶点所构成的三角形可被删除。更坏的情形是，并不是优角顶点链上的所有顶点都是对W可见的。图 13.72 说明了这种构形。在所有有效的三角形被删除之后，W 将是下一个被加入优角顶点链的顶点。

最终的变化是W可能出现在条的上面，而不是里面。在这种情形中，优角顶点链上的所有顶点对 V_0都是可见的。组成所有包含 V_0 和优角顶点的三角形并且删除它们就足够了。应该注意到，在这种情形中，三角形 $\langle V_0, V_{min}, V_1\rangle$ 是多边形的一个耳，刚好就是原来试图首先删除这个三角形的原因。从这个三角形开始删除有效的三角形。图 13.73 说明了这一点。图 13.72 和图 13.73 中的构形的差别是 V_0 和 W的 y 排序。为了处理这种情形，属于左链和右链的顶点必须在一个列表中被排序。由于左链和右链已经被排序，完整的排序要求两个已排序的列表，这是一种执行时间为 $O(n)$的操作。

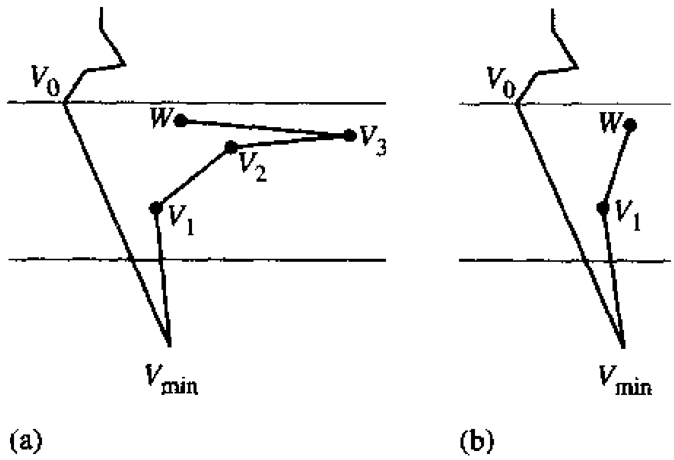


图 13.72　(a) 对 W 来说，并非所有优角顶点链上的顶点都是可见的。
(b) 删除三角形导致 W 成为下一个增加到优角顶点链上的顶点

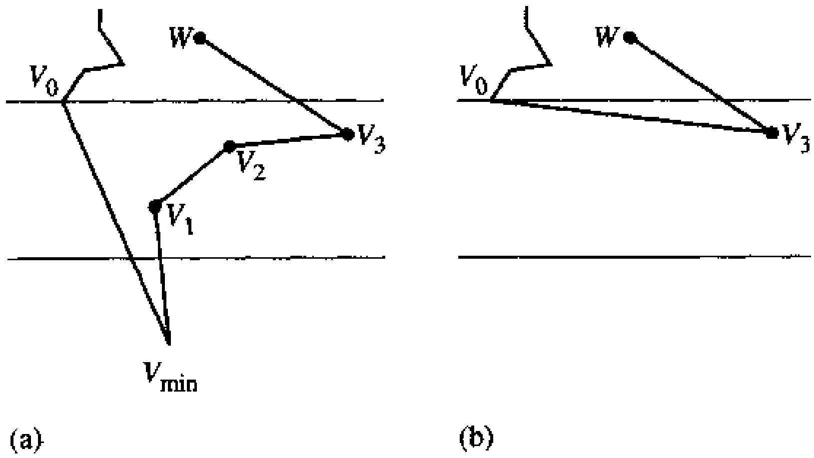


图 13.73　(a) W 出现在当前带的上方，V_0 对所有优角顶点链上的顶点都是可见的。
(b) 删除三角形导致单调多边形的减少，因此处理能重复进行

对 y 单调多边形进行三角剖分的伪码列出如下。左链和右链被分别传入，每一个都具有在第一个槽中的最小 y 值的顶点，并且每一个都具有在对应的最后一个槽中的最大 y 值的顶点。

```
void TriangulateMonotone(int NL, Point LChain[NL], int NR, Point RChain[NR],
                         TriangleList TList)
{
    // Each node in the list contains a vertex and an identifier of the
    // chain to which the vertex belongs.
    VertexList VList = MergeChains(NL, LChain, NR, RChain);

    // A list whose front corresponds to the y-minimum vertex and whose rear
    // corresponds to the y-maximum vertex.
    ReflexChain RList;

    // Initialize the chain with the first two vertices.  AddMax(VList) places
    // the specified list node at the end of the chain.  A side effect of the
```

```
    // operation is to provide RList with a 'whichChain' tag, L or R, about
    // which of the left or right polygon chains the current reflex chain belongs.
    RList.AddMax(VList);  VList = VList.Next();
    RList.AddMax(VList);  VList = VList.Next();

    // VList points to the third vertex in the list.
    while (VList is not empty ) {
        // Max() is an accessor to the rear of the reflex chain that contains
        // the vertex of maximum y-value.
        if (VList.Previous() is equal to RList.Max()) {
            // VList.vertex is on the same chain as the reflex chain
            if (RList.Max() is a convex vertex) {
                TList.Add(RList.Max().Previous().vertex);
                TList.Add(RList.Max().vertex);
                TList.Add(VList.vertex);

                // Remove vertex from reflex chain and from vertex list.  These
                // are the same vertex.
                RList.RemoveMax();
                VList.Previous().RemoveSelf();
                if (RList is empty)
                    VList = VList.Next();
            }  else {
                // RList.Max() is a reflex vertex, no collinear allowed
                RList.AddMax(VList);
                VList = VList.Next();
            }
        } else {
            // VList.vertex is on the opposite chain to the reflex chain.
            // Min() is an accessor to the front of the reflex chain that
            // contains the vertex of minimum y-value.
            TList.Add(RList.Min().vertex);
            TList.Add(RList.Min().Next().vertex);
            TList.Add(VList.vertex);

            // Remove vertex from reflex chain and from vertex list.  These
            // are the same vertex.
            RList.RemoveMin();
            VList.Previous().RemoveSelf();
            if (RList is empty)
                VList = VList.Next();
        }
    }
}
```

概要地说，一个简单多边形的三角剖分可以通过如下的方式来实现：将一个简单多边形分解为梯形的水平条，将梯形合并为最大的条，连接梯形的顶边和底边上的顶点，以构成y单调的多边形，然后三角剖分这个单调多边形。

O'Rourke（1998）提出了这种算法的一种寻找单调山的变体。单调山是其中一条单调

链是一条线段的单调多边形。三角剖分一个单调山比三角剖分一个单调多边形要容易，因为（对于单调山）一次标识和删除一个耳更容易。这种算法的实现看起来与在本章前面介绍过的耳剪裁方法很相似。

13.9.4 凸分解

多边形的三角剖分是将多边形分解为凸子多边形的特殊情形，对于顶点数为n的多边形，分解得到的子多边形数为$n-2$。更一般的问题是将多边形分解为凸子多边形，但是使这样的子多边形数量为最少。显然，三角剖分并没有实现这一点。一个正方形的三角剖分有两个三角形，但已经是凸的，因此凸片的最合适的数量是 1。对于允许艺术家构建三维多边形模型而不在乎产生非凸面的情形，凸分解非常有用。可对这种模型进行后加工，以分解面，使得所有的面都是凸的。而且，对于图元包括凸多边形的着色方法来说，产生最小数量的凸面很有用。通过使输入图元的数量最小化，栅格化多边形的开销也被最小化。

三角剖分总是使用多边形的对角线作为三角形的边。为了获得最小数量的子多边形，最优的凸分解可能要求指定额外的点和线段（如图 13.74 所示）。本节提到的凸分解方法仅在构建中使用对角线。

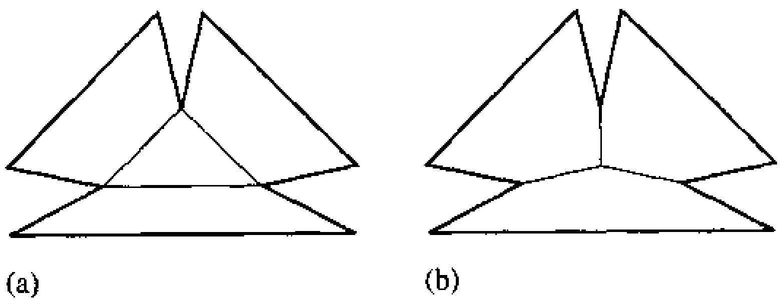


图 13.74　（a）仅使用顶点进行分区。（b）使用多边形内的一个额外点进行分区

Chazelle 证明了在分解中的凸子多边形的最小数量 μ 的范围为 $\lceil r/2\rceil+1\leqslant\mu\leqslant r+1$，其中$r$是多边形的优角顶点的数量。这个范围很窄，因为构建两个多边形很容易，其中的优化分解一个多边形得到较低的限值，另一个的优化分解一个多边形得到较高的限值。构建优化分解的算法一般运行速度很慢（在渐近条件下）。快速的分解算法一般无法获得优化的片数。在本节中，我们各提供一个例子。

1. 一种次优但快速的凸分解

Hertel 和 Mehlhorn（1983）提出了一种快速而简单的次优凸分解算法。然而，已经知道，凸子多边形的数量并不多于优化数量的四倍。给定一个与对角线相关的凸分解，一条对角线与顶点V有关，如果删除两个凸子多边形共用的这条对角线将得到一个不凸的多边形的并集（位于V处），那么这条对角线叫做这个顶点的基本对角线。否则，这条对角线叫做这个顶点的非基本对角线。显然，一条连接两个顶点的对角线是非基本对角线。如果一条对角线对于V是基本对角线，那么这个顶点必须是优角顶点。然而，一个优角顶点可以是一条非基本对角线的一个端点（如图 13.75 所示）。

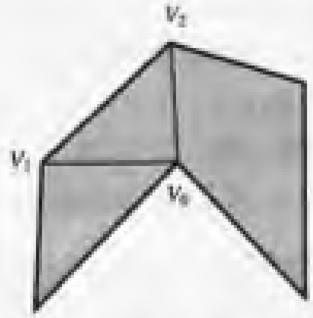


图 13.75 顶点 V_0 是优角顶点。对角线 (V_0, V_1) 是非基本对角线。对角线 (V_0, V_2) 是 V_0 的基本对角线

这个算法很简单。首先三角剖分多边形。删除每一条非基本对角线，一次删除一条。该三角剖分具有 $O(n)$ 条对角线，因此需要删除 $O(n)$ 次。在理论上，三角剖分可在 $O(n)$ 时间内完成，因此次优分解也可在 $O(n)$ 时间内完成。但是正如前面所提到的，三角剖分算法在实际实现时渐近较慢，因此分解所需的时间由三角剖分所需的时间所决定。

2．一种优化凸分解

本节描述了 Keil 和 Snoeyink（1998）提出的一种优化凸分解算法。这种算法仅使用对角线，并且其渐近运行时间是 $O(nr^2)$ 级的，其中 n 是多边形的顶点数量，r 是多边形的优角顶点数量。这种算法以动态编程（Bellman 1987）为基础。

多边形具有逆时针次序的顶点 V_i，其中 $0 \leqslant i < n$。对角线表示为 $d_{ij} = \langle V_i, V_j \rangle$，其中 $i < j$。仅需要考虑那些至少有一个端点是优角顶点的对角线。显然，可以两个端点都是优角顶点的对角线，并将两个凸子多边形并成一个凸子多边形。因此，优化凸分解将不具有两个端点都是优角顶点的对角线。连接两个优角顶点的对角线并不会成为优化分区的一部分。

动态编程通过合并子问题的优化解法来得到一个问题优化解法。给定一条对角线 d_{ik}，子问题与多边形 P_{ik} 相关，这个多边形的顶点为 $V_i, V_{i+1}, \cdots, V_k$。这个多边形本身必须被优化分解为凸片。这个问题的大小就是多边形的顶点数量。原来的多边形为 $P_{0,n-1}$，其大小为 n。子多边形 P_{ik} 具有大小 $k-i+1$。问题的权重 w_{ik} 是 P_{ik} 的凸分解中的对角线数量的最小值。优化分区与计算原始多边形 $P_{0,n-1}$ 的权重 $w_{0,n-1}$ 相关。有一种惯例，即将边 $d_{0,n-1}$ 看成是一条对角线。从底向上进行优化，因此，初始时，我们需要有 $w_{i,i+1} = -1$ 对所有的 i 都成立。

正如 Keil（1985）所指出的，P_{ik} 能具有指数级数量的分解，这些分解都可达到权重 w_{ik}。然而，通过定义分解的等价类，可以简化 P_{ik} 的分解。每一个 P_{ik} 的分解都有一对相关联的顶点索引 $[a, b]$，其中可能 $a = b$，索引为 a, i, k, b 的顶点在分解的凸多边形中按顺时针方向出现。如果两个分解具有相同的权重和相同的关联索引对，那么它们是等价的。此外，一些最小分解被标记为最窄对，那些顶点区域在 d_{ik} 的相邻区域分解不包含任何其他的 P_{ik} 的最小分解的凸区域。Keil（1985）注意到，当构建子问题 P_{ik} 的解时，仅需要考虑最窄对。图 13.76 显示了一个原始多边形（左上角的多边形）和 11 个最小凸分解。最窄对用灰色表示为阴影。正如它所表明的，关于顶点 V_i 的从对角线 d_{ij_1} 到 d_{ij_k} 的逆时针角度次序，与关于多边形的从顶点 V_{j_1} 到顶点 V_{j_k} 的逆时针次序一致。这一观察得出的结论是，子问题 P_{ik} 的

最窄对可通过比较相关联的对的索引来检测。只要遇到另一个具有较少索引的相关联对 $[a_1, b_1]$，则可能的分解的任何相关联对 $[a_0, b_0]$ 都被抛弃。如果 $a_1 \leqslant a_0$，则必定出现 $b_1 \leqslant b_0$ 的情形。随着子问题 P_{ik} 的最窄对被计算出来，它们被压入一个栈中，使得栈中从底到顶的对是关于顶点 V_i 和 V_k 按逆时针次序排列的。图 13.76 中的多边形堆栈将包含[1, 3]，[3, 4]和[6, 8]，最后一对为栈顶。对角线 d_{06} 和 d_{69} 因而构成逆时针方向上最远的最窄对。

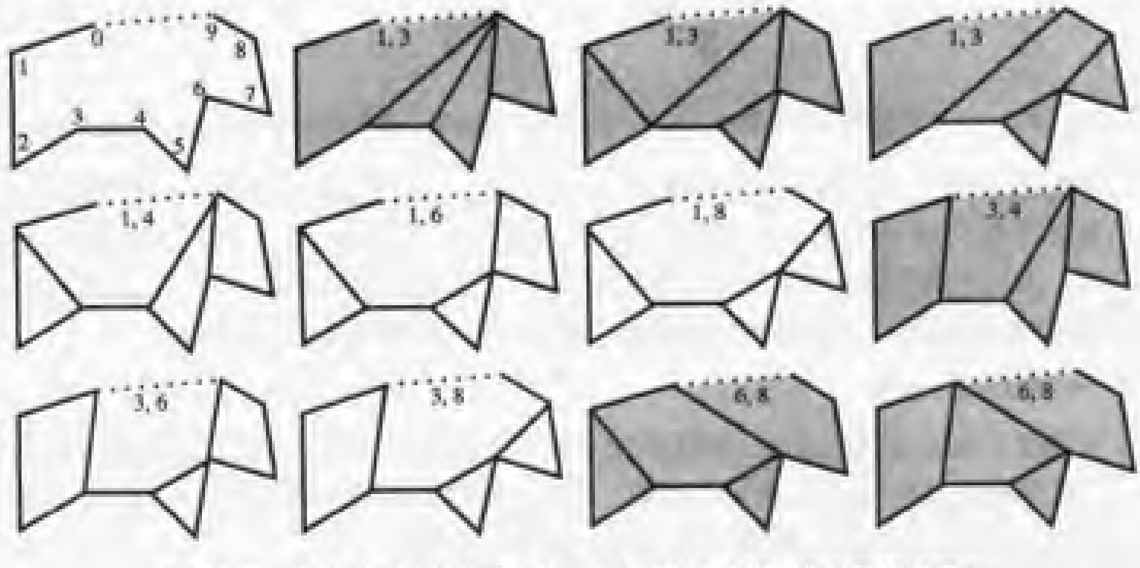


图 13.76　原多边形（左上角）和 11 个最小凸分解，其中最窄对用灰色显示。点线表示作为多边形的边而不是对角线来处理

另一个关键思想与被分解的凸多边形的规范三角剖分相关。凸多边形的每一个三角剖分都是一个三角形扇，其基顶点为该多边形的一个优角顶点。图 13.77 表示了这种规范三角剖分。图中的多边形摘自 Keil 和 Snoeyink（1998），但是其中的顶点标记不一样，以满足该论文中的限制条件，即 V_0 是一个优角顶点。作为扇形的基顶点的优角顶点在这个凸多边形的所有优角顶点中具有最小的索引，扇形在这个凸多边形中构建。Keil 和 Snoeyink 提到了如下的事实。在凸多边形 P 的规范三角剖分中，每一个具有 $i < k$ 的对角线 d_{ik} 都满足三个与子多边形 P_{ik} 相关的条件：

（1）端点在 P_{ik} 上的对角线定义 P_{ik} 的一个规范三角剖分。

（2）如果 V_i 是 P 的一个优角顶点，那么，对于三角形 $\langle V_i, V_j, V_k\rangle$，其中 $i < j < k$，有 $j = k - 1$ 或者 d_{jk} 是一条用于凸分解的对角线。

（3）如果 V_i 不是 P 的一个优角顶点，那么 V_k 是一个优角顶点，对于三角形 $\langle V_i, V_j, V_k\rangle$，其中 $i < j < k$，有 $j = i + 1$ 或者 d_{ij} 是一条用于凸分解的对角线。

在图 13.77 中，考虑子多边形 $P_{9,11}$，使其满足 $i = 9$ 和 $k = 11$。顶点 V_9 是优角顶点，因此可以应用条件 2。三角形 $\langle V_9, V_{10}, V_{11}\rangle$ 位于 P 的规范三角剖分中，并具有 $j = 10 = k - 1$。在子多边形 $P_{16,23}$ 中，$i = 16$ 且 $k = 23$。顶点 V_{16} 是优角顶点，因此可应用条件 2。三角形 $\langle V_{16}, V_{17}, V_{23}\rangle$ 位于 P 的规范三角剖分中，但是 $j = 17 \neq 22 = k - 1$。然而，$d_{17,23}$ 是用于图分解的对角线。考虑子多边形 $P_{5,9}$，使其满足 $i = 5$ 和 $k = 9$。顶点 V_5 不是一个优角顶点，但是 V_9 是一个优角顶点。三角形 $\langle V_5, V_6, V_9\rangle$ 位于 P 的规范三角剖分中，并具有 $j = 6 = i + 1$。

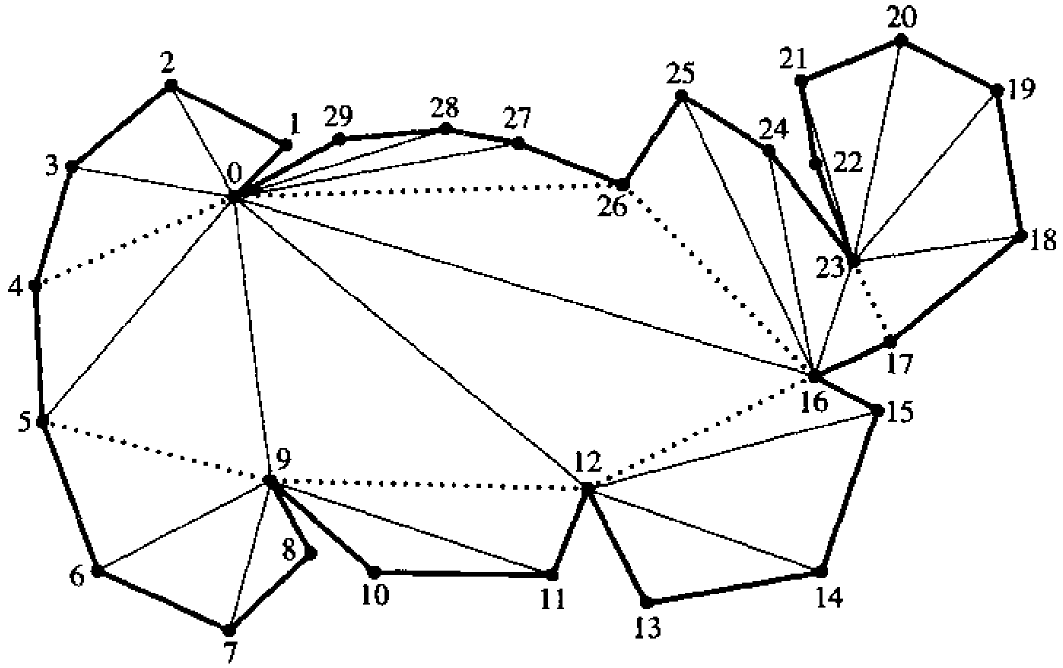


图 13.77 多边形的最小凸分解中使用的凸多边形的规范三角剖分。原多边形的边用粗线表示。分解中使用的对角线用点线表示。用于规范三角剖分三角扇的对角线用细线表示

通过考虑哪些顶点 V_j 构成与对角线 P_{ik} 有关的规范三角剖分，以及对角线 d_{ij} 和 d_{jk} 是 P_{ij} 和 P_{jk} 的分解的一部分还是仅仅是规范三角剖分的一部分，就能构建 P_{ik} 的最小凸分解。将它们组合在一起，这个算法用于检验 P_{ik} 的具有最窄对的最小分解的规范三角剖分。这个算法将堆栈 S_{ik} 和 T_{ij} 与每一个子多边形 P_{ik} 关联在一起，前一个堆栈按递增次序存储 P_{ik} 的最窄对，后一个堆栈按递减次序存储这些最窄对。这种算法的一种变体是，在分析 P_{ik} 时，每一个小于 P_{ij} 的子问题 P_{xy} 都具有其中所有的最窄对，保存在对应的堆栈中。一次只对一个子问题使用这两个堆栈中的一个，因此它们都可在内存中重叠。用于 S_{ik} 的数据结构因而就是一个“双端栈”。对 S_{ik} 的操作应用于堆栈的一端，而对 T_{ik} 的操作应用于堆栈的另一端。

考虑上面的条件 2，其中 V_i 是一个优角顶点。P_{ik} 的最小分解使用对角线（或者边）d_{jk}（其中 j 严格地位于 i 和 k 之间）、P_{ik} 的一个分解和 P_{ij} 的一个分解，其中最后一个分解不一定包含 d_{ij}。动态编程的循环逻辑为

$$w_{ik}=\min_{i<j<k,\, d_{ij}\text{ 和 } d_{jk}\text{ 存在}}\begin{cases} w_{ij}+w_{jk}+2, & \text{如果 } d_{ij} \text{ 包含于分解中} \\ w_{ij}+w_{jk}+1, & \text{否则} \end{cases}$$

对于 j 的一个值，从堆栈 T_{ij} 中取出值将返回逆时针次序的对。搜索最后的一个使得 d_{ij} 和 d_{jk} 并不在 V_j 处构成一个优角的对 $[s,t]$。如果不存在这样的对，或者 d_{is} 和 d_{ik} 在 V_i 处形成一个优角，那么在 P_{ik} 的凸分解中需要 d_{ij}，并具有权重 $w_{ij}+w_{ik}+2$ 和最窄对 $[j,j]$。否则，P_{ik} 的一个凸分解就已经找到了其权重 $w_{ij}+w_{jk}+1$ 和最窄对 $[s,j]$。对于所选择的 j，在 S_{ik} 上的对的第二个索引总为 j。只有当栈顶 $[x_0,j]$ 满足 $x_0<x$ 时，通过将每一个达到了最小权重的对 $[x,j]$ 压入堆栈，就能构建堆栈。如果 $x_0\geqslant x$，那么栈顶比候选对 $[x,j]$ 更窄，因此它不被压入。对角线 d_{jk} 用于分解，因此 $j=k-1$（对角线实际上是一条多边形

边）或者至少 V_j 或 V_k 中的一个是优角顶点。在 Keil 和 Snoeyink（1998）中，这种条件叫做“类型 A”，因此伪码函数就采用这个名称。双端堆栈表示为 S(i,j) 或者 T(i,j)，取决于操作是应用于堆栈的哪一端。对角线表示为 D(i,j)。用于跟踪记录最窄对的索引对为 (pair.first,pair.second)。

```
void TypeA(int i, int j, int k)
{
    pair = null;
    while (T(i,j) is not empty) {
        tmpPair = T(i,j).Pop();
        if (D(tmpPair.second,j ) and D(j, k) are not reflex at j)
            pair = tmpPair;
    }

    if ((pair == null) or (D(i,pair.first) and D(i,k) are reflex at i)) {
        P(i, k) decomposition uses D(i,j );
        wtmp = w(i, j) + w(j, k) + 2;
        narrow = [j, j];
    } else {
        P(i, k) decomposition does not use D(i, j);
        wtmp = w(i, j) + w(j, k) + 1;
        narrow = [pair.first, j];
    }

    if (S(i, k) is empty) {
        w(i, k) = wtmp;
        S(i, k).Push(narrow);
    }  else if (wtmp < w(i, k)) {
        S(i, k).PopAll();
        w(i, k) = wtmp;
        S(i, k).Push(narrow);
    }  else if (wtmp == w(i, k)) {
        if (narrow.first > S(i, k).Top().first)
            S(i, k).Push(narrow);
    }
}
```

除检测顶点 V_i 和 V_k 是否优角顶点外，条件 3 是条件 2 的均衡情形，在这种次序中，V_i 是一个凸角顶点，而 V_k 是一个优角顶点。两个顶点都是优角顶点的情形可被条件 2 所捕获。在 Keil 和 Snoeyink（1998）中，这种条件被称为“类型 B”，因此伪码函数就采用这个名称。

```
void TypeB(int i, int j, int k)
{
    pair = null;
    while (S(j,k) is not empty) {
        tmpPair = S(j, k).Pop();
        if (D(i, j) and D(j, tmpPair.first) are not reflex at j)
            pair = tmpPair;
    }
```

```
        if ((pair == null) or (D(pair.second, k) and D(i, k) are reflex at k)) {
            P(i, k) decomposition uses D(j, k);
            wtmp = w(i, j) + w(j, k) + 2;
            narrow = [j, j];
        } else  {
            P(i, k) decomposition does not use D(j, k);
            wtmp = w(i, j) + w(j, k) + 1;
            narrow = [j, pair.second];
        }

        if (S(i, k) is empty) {
            w(i, k) = wtmp;
            S(i, k).Push(narrow);
        } else if (wtmp < w(i, k)) {
            S(i, k).PopAll();
            w(i, k) = wtmp;
            S(i, k).Push(narrow);
        }  else if (wtmp == w(i, k)) {
            while (narrow.second <= S(i, k).Top().second )
                S(i, k).Push(narrow);
        }
    }
```

下面列出了最小凸分解（MCD）的主函数。逆时针次序的顶点V[n]被传入该函数。优角顶点RV[r]也被传入，以保持算法的 $O(nr^2)$ 时间级。预处理这些优角顶点的时间级为 $O(n)$。

```
void MCD(int n, Point V[n], int r, Point RV[r])
{
    // size 2 problems
    for (i = 0, k = 1; k < n; i++, k++)
        w(i, k) = -1;
    // size 3 problems
    for (i = 0, k = 2; k < n; k++) {
        if (Visible(i, k)) {
            w(i, k) = 0;
            S(i, k).Push([i + 1, i + 1]);
        }
    }

    // size 4 and larger problems
    for (size = 4; size <= n; size++) {
        for (m = 0; m < r; m++) {
            i = RV[m];  k = i + size - 1;
            if (k >= n) break;
            if (Visible(i, k)) {
                if (Reflex(k)) {
                    for (j = i + 1; j <= k - 1; j++) {
                        if (Visible(i, j) and Visible(j, k))
                            TypeA(i, j, k);
                    }
```

```
                } else {
                    for (j = i + 1; j <= k - 2; j++) {
                        if (Reflex(j) and Visible(i, j) and Visible(j, k))
                            TypeA(i, j, k);
                    }
                    if (Visible(i, k - 1))
                        TypeA(i, k - 1, k);
                }
            }
        }

        for (m = r - 1; m >= 0; m--) {
            k = RV[m];  i = k - size + 1;
            if (i < 0) break;
            if ((not Reflex(i)) and Visible(i, k)) {
                if (Visible(i + 1, k))
                    TypeB(i, i + 1, k);
                for (j = i + 2; j <= k - 1; j++) {
                    if (Reflex(j) and Visible(i, j) and Visible(j, k))
                        TypeB(i, j, k);
                }
            }
        }
    }
}
```

在大小为 4 或者更大的代码块中，该函数的时间是 $O(nr^2)$ 级的，因为仅对至少两个优角顶点或者一个优角顶点和一条多边形边调用 TypeA 和 TypeB 。每一次调用 TypeA 和 TypeB 的时间级都是 $O(1)$ 加上从堆栈中取出的对的数量。由于至多一个对被加入两个堆栈中，因此最多可从堆栈中取出 $O(nr^2)$ 个元素。根据堆栈所要求的空间，所需要的内存也是 $O(nr^2)$ 级的。

乍一看来，大小为 3 的代码块似乎是 $O(n^2)$ 时间级的，即 $O(n)$ 用于外层循环，$O(n)$ 用于每一次确定 $\langle V_i, V_{i+2}\rangle$ 是否对角线的可见性检测。对于大小为 4 或者更大的块，这样可能改变时间级 $O(nr^2)$，其中 r 比 n 小得多。由于大小为 3，因此可见性检测其实就是检查 $\langle V_i, V_{i+2}\rangle$ 是否是一个耳。正如在本节的开头所说明的，可以通过只有优角顶点位于三角形 $\langle V_i, V_{i+1}, V_{i+2}\rangle$ 内的包含性检测来实现耳的检测。因此，实现大小为 2 的块的时间级为 $O(nr)$。

3. 杂项

也可以使用空间分区二叉树来实现多边形的分解。在 13.1 节中已经介绍了处理多边形的空间分区二叉树的计算方法。树的叶子节点表示平面的一个凸分解。正 / 负标记允许我们确定对应于原始多边形的凸子多边形的叶子节点。它们的并就是原始的多边形。这种类型的分解将点插入多边形，这与前一节讨论的仅适用于原始顶点的方法不一样。如果需要一个多边形的三角剖分，那么凸子多边形可被扇形化为三角形。

将多面体分解为四面体的问题是平面多边形分解的自然扩展。到目前为止，Chazelle 和 Palios（1990）所提出的算法是最快的对非凸多面体进行三角剖分的算法。其渐近级别是 $O(n\log r + r^2\log r)$，其中 n 是面的数量，r 是优角边的数量。另一篇相关的论文是 Hershberger 和 Snoeyink（1997），其中介绍了使用空间分区二叉树来将非凸多面体分解为凸片的有效方法。在实际应用中，空间分区二叉树的实现很容易，并且这种方法所得到的分解性能是可以接受的。

13.10　外接球与内切球

有两个特殊的圆与二维空间中的三角形相关，即一个包含三角形的三个顶点的外接圆和一个包含于三角形内且面积最大的内切圆。虽然内切圆是所有包含于三角形内的圆中面积最大的圆，然而，外接圆却不一定是所有包含三角形的圆中面积最小的圆。一种很明显的情形是，但三角形的顶点几乎共线时，外接圆具有非常大的面积，但是，包含三角形的最小面积的圆其直径等于最长的边的长度。图 13.78 说明了一个三角形的外接圆和内切圆。外接圆用实线表示，而内切圆用点线表示。我们的目的是构建一个指定三角形的外接圆和内切圆。

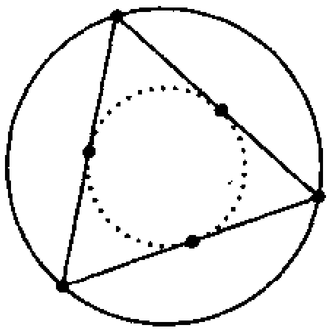


图 13.78　三角形的外接圆和内切圆

与之类似，有两个特殊的球与三维空间中的四面体相关，即一个包含四面体的顶点的外接球和一个包含于四面体内的体积最大的内切球。外接球不一定是所有包含四面体的球中体积最小的球。

这种思想可以扩展到更高维空间。三角形（二维空间）和四面体（三维空间）在 n 维空间中的一般化叫做单形。该对象具有 $n+1$ 个顶点，每一个顶点与其他的每一个顶点都相连。如果顶点为 V_i（$0 \leqslant i \leqslant n$），那么边 $\vec{e}_i = V_i - V_0$ 必须是线性无关的向量。为了说明这种限定条件的意义，在三维空间中，该条件排除了四个点共面（即四面体是平的且没有体积）的情形。一个单形的两种特殊的超球（数学术语）或者球（通俗名称）为一个包含单形的顶点的外接球和一个包含于单形内的体积最大的内切球。

构建外接球和内切球与建立具有 n 个未知数的 n 个线性方程相关。由于这种构建的简单性，并不需要分别处理二维空间和三维空间的情形以提供直观知识。

13.10.1 外接球

单形的外接球就是通过单形的所有顶点的球。球的球心C与所有的顶点之间的距离都相同，设该距离为r。限定条件为

$$\|C - V_i\| = r,\quad 0 \leqslant i \leqslant n$$

对该距离平方，扩展其中的点积，并减去$i = 0$时的平方后方程，可得$2(V_i - V_0)\cdot(C - V_0) - \|V_i - V_0\|^2$，其中$1 \leqslant i \leqslant n$。这是一个形式为$\mathrm{A}X = B$的线性方程系统，其中A的第$i$行为$V_i - V_0$，表示为一个$1\times n$的向量，$B$的第$i$行为$\|V_i - V_0\|^2/2$，而$X = C - V_0$表示为一个$n\times 1$的向量。由于共用$V_0$的边都是线性无关的，因此A是一个可逆矩阵，并且该线性系统具有惟一的解$X = \mathrm{A}^{-1}B$。因此，外接球的球心为$C = V_0 + \mathrm{A}^{-1}B$。一旦求得了外接球的球心，就可计算出其半径为$r = \|C - V_0\|$。

1．二维空间

定义$V_i = (x_i, y_i)$，其中i=0, 1, 2。假设三角形是逆时针次序的。定义$X_i = x_i - x_0$和$Y_i = y_i - y_0$。三角形的面积为

$$A = \frac{1}{2}\det\begin{bmatrix} X_1 & Y_1 \\ X_2 & Y_2 \end{bmatrix}$$

中心(x, y)和半径r为

$$x = x_0 + \frac{1}{4A}(Y_2L_{10}^2 - Y_1L_{20}^2)$$

$$y = y_0 + \frac{1}{4A}(X_1L_{20}^2 - X_2L_{10}^2)$$

$$r = \sqrt{(x - x_0)^2 + (y - y_0)^2}$$

其中$L_{ij} = \|V_i - V_j\|$。可以证明$r = L_{10}L_{20}L_{12}/(4A)$，即边长之积除以面积的 4 倍。还可以证明（Blumenthal 1970）半径是 Cayley-Menger 行列式方程的一个解

$$\det\begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & L_{10}^2 & L_{20}^2 & r^2 \\ 1 & L_{10}^2 & 0 & L_{21}^2 & r^2 \\ 1 & L_{20}^2 & L_{21}^2 & 0 & r^2 \\ 1 & r^2 & r^2 & r^2 & 0 \end{bmatrix} = 0$$

2．三维空间

定义$V_i = (x_i, y_i, z_i)$，其中$i = 0, 1, 2, 3$。假设四面体(V_0, V_1, V_2, V_3)的次序与规范次序$((0,0,0),(1,0,0),(0,1,0),(0,0,1))$是同构的。定义$X_i = x_i - x_0$, $Y_i = y_i - y_0$和$Z_i = z_i - z_0$。四面体的体积为

$$V = \frac{1}{6}\det\begin{bmatrix} X_1 & Y_1 & Z_1 \\ X_2 & Y_2 & Z_2 \\ X_3 & Y_3 & Z_3 \end{bmatrix}$$

中心(x,y,z)和半径r为

$$x = x_0 + \frac{1}{12V}\left(+(Y_2Z_3 - Y_3Z_2)L_{10}^2 - (Y_1Z_3 - Y_3Z_1)L_{20}^2 + (Y_1Z_2 - Y_2Z_1)L_{30}^2\right)$$

$$y = y_0 + \frac{1}{12V}\left(-(X_2Z_3 - X_3Z_2)L_{10}^2 + (X_1Z_3 - X_3Z_1)L_{20}^2 - (X_1Z_2 - X_2Z_1)L_{30}^2\right)$$

$$z = z_0 + \frac{1}{12V}\left(+(X_2Y_3 - X_3Y_2)L_{10}^2 - (X_1Y_3 - X_3Y_1)L_{20}^2 + (X_1Y_2 - X_2Y_1)L_{30}^2\right)$$

$$r = \sqrt{(x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2}$$

其中$L_{ij} = \|V_i - V_j\|$。可以证明（Blumenthal 1970）半径是 Cayley-Menger 行列式方程的一个解

$$\det\begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & L_{10}^2 & L_{20}^2 & L_{30}^2 & r^2 \\ 1 & L_{10}^2 & 0 & L_{21}^2 & L_{31}^2 & r^2 \\ 1 & L_{20}^2 & L_{21}^2 & 0 & L_{32}^2 & r^2 \\ 1 & L_{30}^2 & L_{31}^2 & L_{32}^2 & 0 & r^2 \\ 1 & r^2 & r^2 & r^2 & r^2 & 0 \end{bmatrix} = 0$$

13.10.2 内切球

单形的内切球就是包含于单形内的体积最大的球。这个球必须与这个单形的所有面相切。球的球心C与所有的面之间的距离都相同，设该距离为r。球心与每一个面的距离可通过$C-V$在包含V的面的指向内的单位长度法线向量上的投影长度来获得。该投影为

$$\hat{n}_i \cdot (C - V_i) = r, \quad 0 \leqslant i \leqslant n$$

其中$\hat{n}_i$为由顶点$V_{i\bmod(n+1)}, V_{(i+1)\bmod(n+1)}, \cdots, V_{(i+n-1)\bmod(n+1)}$所确定的超面的指向内的单位长度法线。这是一个具有$n+1$个未知数$(C,r)$的$n+1$个方程的线性系统，其中每一个方程都表示为$(\hat{n}_i,-1)\cdot(C,r) = \hat{n}_i \cdot V_i$。定义$(n+1)\times(n+1)$的矩阵$\mathbf{A}$，其第$i$行为向量$(\hat{n}_i,-1)$，表示为向量$1\times(n+1)$。定义$(n+1)\times 1$的向量$B$，其第$i$行为$\hat{n}_i \cdot V_i$。线性系统为$\mathbf{A}(C,r) = B$，并且具有解$(C,r) = \mathbf{A}^{-1}B$，其左边可看成是一个（$n+1$）×1 向量。

1. 二维空间

定义$V_i = (x_i, y_i)$，其中$i = 0,1,2$。为了表示起来方便，设$V_3 = V_0$。单位长度边的方向为$\hat{d}_i = (V_{i+1} - V_i)/L_i$，其中$L_i = \|V_{i+1} - V_i\|$，其中 $0 \leqslant i \leqslant 2$。指向内的单位长度法线为$\hat{n}_i = -\hat{d}_i^{\perp}$，其中$(x,y)^{\perp} = (y,-x)$。

内切圆的圆心和半径可用上述的方法来构建。然而，如果圆心用重心坐标表示为$C = t_0V_0 + t_1V_1 + t_2V_2$，其中$t_0 + t_1 + t_2 = 1$，那么其解对于圆心将具有很好的对称性。在这种形式中，方程$r = \hat{n}_i \cdot (C - V_i)$成为$r = t_2L_2\hat{d}_0 \cdot \hat{n}_2$, $r = t_0L_0\hat{d}_1 \cdot \hat{n}_0$和$r = t_1L_1\hat{d}_2 \cdot \hat{n}_1$。三角形的面积$A$由$2A = L_0L_2\hat{d}_0 \cdot \hat{n}_2 = L_1L_0\hat{d}_1 \cdot \hat{n}_0 = L_2L_1\hat{d}_2 \cdot \hat{n}_1$给出。将它们与上述的方程组合在一起，可得$t_0 = RL_1/(2A), t_1 = RL_2/(2A)$和$t_2 = RL_0/(2A)$。将$t_i$相加，可得$1 = (L_0 + L_1 + L_2)r/(2A)$，其中$r = 2A/(L_0 + L_1 + L_2)$。为了表示起来方便，定义$\ell_i = L_{(i-1)\bmod 3}$。值$\ell_i$是顶

点V_i所对的边的长度。定义$L = L_0 + L_1 + L_2 = \ell_0 + \ell_1 + \ell_2$。在这种形式中，半径和圆心为

$$r = \frac{2A}{L}, \quad C = \sum_{i=0}^{2} \frac{\ell_i}{L} V_i$$

2. 三维空间

定义$V_i = (x_i, y_i, z_i)$，其中$i = 0, 1, 2, 3$。四面体$\langle V_0, V_1, V_2, V_3 \rangle$的次序与规范次序$\langle (0,0,0), (1,0,0), (0,1,0), (0,0,1) \rangle$是同构的。该构形的指向内的法线为$\vec{n}_0 = (V_1 - V_0) \times (V_2 - V_0)/(2A_0)$，其中$A_0$是以$\vec{n}_0$为法线的面的面积；$\vec{n}_1 = (V_3 - V_1) \times (V_2 - V_1)/(2A_1)$，其中$A_1$是以$\vec{n}_1$为法线的面的面积；$\vec{n}_2 = (V_3 - V_2) \times (V_0 - V_2)/(2A_2)$，其中$A_2$是以$\vec{n}_2$为法线的面的面积；以及$\vec{n}_3 = (V_1 - V_3) \times (V_2 - V_3)/(2A_3)$，其中$A_3$是以$\vec{n}_3$为法线的面的面积。

与在二维空间中一样，当球心用重心坐标表示为$C = \sum_{i=0}^{3} t_i V_i$，其中$\sum_{i=0}^{3} t_i = 1$时，其解对于球心将具有很好的对称性。方程$r = \vec{n}_i \cdot (C - V_i)$成为$r = t_3 \vec{n}_0 \cdot (V_3 - V_0), r = t_0 \vec{n}_1 \cdot (V_0 - V_1), r = t_1 \vec{n}_2 \cdot (V_1 - V_2)$和$r = t_2 \vec{n}_3 \cdot (V_2 - V_3)$。四面体的体积由下面与三元数量积相关的方程给出：$6V = [V_1 - V_0, V_2 - V_0, V_3 - V_0] = [V_0 - V_1, V_3 - V_1, V_2 - V_1] = [V_3 - V_2, V_0 - V_2, V_1 - V_2] = [V_2 - V_3, V_1 - V_3, V_0 - V_3]$。将它们与上述的方程组合在一起，可得$t_0 = RA_1/(3V), t_1 = RA_2/(3V), t_2 = RA_3/(3V)$和$t_3 = RA_0/(3V)$。将$t_i$相加，可得$1 = (A_0 + A_1 + A_2 + A_3)r/(3V)$，其中$r = 3V/(A_0 + A_1 + A_2 + A_3)$。定义$\alpha_i = A_{(i-1)\bmod 4}$。值$\alpha_i$是顶点$V_i$所对的面的面积。定义$A = A_0 + A_1 + A_2 + A_3 = \alpha_0 + \alpha_1 + \alpha_2 + \alpha_3$。在这种形式中，半径和球心为

$$r = \frac{3V}{A}, \quad C = \sum_{i=0}^{3} \frac{\alpha_i}{A} V_i$$

3. n维空间

用重心坐标来表示C的相同构建方法可应用于一般维空间。半径和球心为

$$r = \frac{nV}{S}, \quad C = \sum_{i=0}^{n} \frac{\sigma_i}{S} V_i$$

其中V是单形的体积，σ_i为顶点V_i所对的超面的曲面面积，而$S = \sum_{i=0}^{n} \sigma_i$是单形的总曲面面积。

13.11　点集的最小区域

在本节中，设点集为$\{P_i\}_{i=0}^{n-1}$，其中$n \geqslant 2$。在本节的讨论中，假设所有点都是惟一的。然而，必须有一种实现能够处理包含同一个点的多份拷贝，或者甚至是近似于同一个点的两个点（在某些浮点容差之内）的点集。本节涵盖了二维空间中的最小面积的矩形、圆和椭圆，以及三维空间中的最小体积的箱子、球和椭球等内容。

13.11.1　最小面积矩形

显然，仅需要考虑原始点集的凸包的顶点。因此，这个问题简化为寻找包含一个凸多

边形的最小面积的矩形，该多边形的有序顶点为P_i，其中$0 \leq i < N$。该矩形并不需要与坐标轴对齐。凸多边形至少有一条边必须包含于最小面积矩形的一条边内。假设满足这一条件，计算最小面积矩形的算法就仅需要计算由多边形的边来确定方向的最紧缩的有界矩形。

1．边的包含性的证明

用反证法来证明边的包含性。假设凸多边形的任何边都不被最小面积矩形的边所包含。该矩形必须由凸多边形的两个、三个或者四个顶点所支持。图 13.79 说明了四个支持顶点的情形。支持顶点用黑色画出，并标记为V_0至V_3。其他的多边形顶点用白色表示。为了便于讨论，旋转该凸多边形，使得矩形的轴为（1，0）和（0，1）。

定义$\vec{u}_0(\theta) = (\cos\theta, \sin\theta)$和$\vec{u}_1(\theta) = (-\sin\theta, \cos\theta)$。存在一个值$\varepsilon > 0$使得$V_i$总是有界矩形的一个支持顶点，其中轴$\vec{u}_0(\theta)$和$\vec{u}_1(\theta)$对所有$\theta$都满足$|\theta| <= \varepsilon$。为了计算有界矩形的面积，将支持顶点投影到轴线$V_0 + s\vec{u}_0(\theta)$和$V_0 + t\vec{u}_1(\theta)$上。投影区间为$[0, s_1]$和$[t_0, t_1]$，其中$s_1 = \vec{u}_0(\theta)\cdot(V_2 - V_0), t_0 = \vec{u}_1(\theta)\cdot(V_1 - V_0)$和$t_1 = \vec{u}_1(\theta)\cdot(V_3 - V_0)$。

定义$\vec{k}_0 = (x_0, y_0) = V_2 - V_0$和$\vec{k}_1 = (x_1, y_1) = V_3 - V_1$。从图 13.79 中可以清楚地看出$x_0 > 0$和$y_1 > 0$。对于$|\theta| \leq \varepsilon$，该矩形的面积为

$$A(\theta) = s_1(t_1 - t_0) = [\vec{k}_0 \cdot \vec{u}_0(\theta)][\vec{k}_1 \cdot \vec{u}_1(\theta)]$$

特别地，$A(0) = x_0 y_1 > 0$。

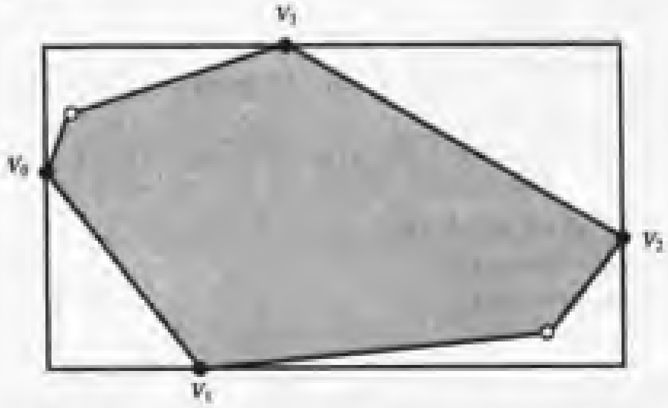


图 13.79　没有一致的多边形边的假设的最小面积矩形

由于$A(\theta)$在其定义域上是可微的，并且假设$A(0)$为全局最小值，因此必有$A'(0) = 0$。一般地，

$$\begin{aligned} A'(\theta) &= [\vec{k}_0 \cdot \vec{u}_0(\theta)][\vec{k}_1 \cdot \vec{u}_1'(\theta)] + [\vec{k}_0 \cdot \vec{u}_0'(\theta)][\vec{k}_1 \cdot \vec{u}_1(\theta)] \\ &= -[\vec{k}_0 \cdot \vec{u}_0(\theta)][\vec{k}_1 \cdot \vec{u}_0(\theta)] + [\vec{k}_0 \cdot \vec{u}_1(\theta)][\vec{k}_1 \cdot \vec{u}_1(\theta)] \end{aligned}$$

因此，$0 = A'(0) = -x_0 x_1 + y_0 y_1$或者$x_0 x_1 = y_0 y_1$。由于$x_0 > 0$且$y_1 > 0$，因此必有$\mathrm{Sign}(x_1) = \mathrm{Sign}(y_0)$。而且，由于假设$A(0)$为全局最小值，因此必定有$A''(0) \geq 0$。一般地，

$$\begin{aligned} A''(\theta) = &-[\vec{k}_0 \cdot \vec{u}_0(\theta)][\vec{k}_1 \cdot \vec{u}_0'(\theta)] - [\vec{k}_0 \cdot \vec{u}_0'(\theta)][\vec{k}_1 \cdot \vec{u}_0(\theta)] \\ &+ [\vec{k}_0 \cdot \vec{u}_1(\theta)][\vec{k}_1 \cdot \vec{u}_1'(\theta)] + [\vec{k}_0 \cdot \vec{u}_1'(\theta)][\vec{k}_1 \cdot \vec{u}_1(\theta)] \end{aligned}$$

$$= -[\vec{k}_0 \cdot \vec{u}_0(\theta)][\vec{k}_1 \cdot \vec{u}_1(\theta)] - [\vec{k}_0 \cdot \vec{u}_1(\theta)][\vec{k}_1 \cdot \vec{u}_0(\theta)]$$

$$- [\vec{k}_0 \cdot \vec{u}_1(\theta)][\vec{k}_1 \cdot \vec{u}_0(\theta)] - [\vec{k}_0 \cdot \vec{u}_0(\theta)][\vec{k}_1 \cdot \vec{u}_1(\theta)]$$

$$= -2\left\{[\vec{k}_0 \cdot \vec{u}_0(\theta)][\vec{k}_1 \cdot \vec{u}_1(\theta)] + [\vec{k}_0 \cdot \vec{u}_1(\theta)][\vec{k}_1 \cdot \vec{u}_0(\theta)]\right\}$$

特别地，$A''(0) = -2(x_0y_1 + x_1y_0) \geqslant 0$。然而，注意由于$A(0) > 0$，因此$x_0y_1 > 0$，以及由于$\mathrm{Sign}(x_1) = \mathrm{Sign}(y_0)$，因此$x_1y_0 > 0$，这就说明$A''(0) < 0$，与我们最初的假设相矛盾。

2. 一种改进

由于最小面积矩形必须包含一条边，因此，简单的实现仅需对凸包的边进行迭代。对于每一条边，计算对应的由这条边所定义的方向上的最小矩形。计算所有这些矩形的最小面积。这种算法的伪码为

```
ordered vertices P[0] through P[N - 1];
define P[N] = P[0];

minimumArea = infinity;
for (i = 1; i <= N; i++)  {
    U0 = P[i] - P[i - 1];
    U1 = (-U0.x, U0.y);
    s0 = t0 = s1 = t1 = 0;
    for (j = 1; j < N; j++) {
        D = P[j] - P[0];
        test = Dot(U0, D);
        if (test < s0) s0 = test; else if (test > s1) s1 = test;
        test = Dot(U1, D);
        if (test < t0) t0 = test; else if (test > t1) t1 = test;
    }
    area = (s1 - s0) * (t1 - t0);
    if (area < minimumArea)
        minimumArea = area;
}
```

这种算法是$O(n^2)$时间级的，因为它有两重循环，每一个循环需要迭代n个项目。下一节将考虑一个更好的算法，这种算法以所谓的旋转测径法为基础。

3. 旋转测径

旋转测径法是 Michael Shamos 的论文（1978）中的思想，这篇论文被认为是计算几何领域的奠基之作。这篇论文中的算法使用了用于计算凸多边形的直径的$O(n)$时间级的方法。Godfreid Toussaint 将其称为“旋转测径”，因为这种方法类似于旋转一对环绕多边形的测径器。在求解其他问题时，这种方法很有用。这些问题包括计算两个凸多边形之间的最小和最大距离，洋葱形的三角剖分（这对出现在一个函数$f(x, y)$的不同等高线上的点集的三角剖分非常有用），合并凸包，凸多边形的相交，以及计算两个凸多边形的闵可夫斯基和／差（与本书的 6.10 节做一比较）。旋转测径的主页（Pirzadeh 1999）中包含了上述算法和其他更多算法的简介。

利用旋转测径来寻找包含一个凸多边形的最小面积矩形很简单。选择该多边形的一条

初始边。该边的方向和垂直方向用于寻找该方向上的最小有界矩形。在构造时跟踪支持该矩形的顶点和边。一次旋转该矩形的一条边。开始于一个支持顶点的边与包含该顶点的矩形的边构成一个角。用所有支持顶点的角中的最小角旋转矩形。新的支持边不需要与前面的支持边相邻。对于新的方向，矩形的大小可在$O(1)$时间内更新。该多边形具有n条边需要访问，对于矩形的每一次旋转，更新时间都是$O(1)$，因此算法的总时间为$O(n)$。

13.11.2 最小体积箱体

与二维空间中的问题一样，仅需要考虑原始点集的凸包的顶点。因此，这个问题简化为寻找包含一个凸多面体的最小体积的箱体，O'Rourke（1985）证明了一个箱面必须包含一个多面体面，而且另一个箱面必须包含一条多面体边；或者三个箱面必须分别包含一个多面体面。第一种情形可在$O(n^2)$时间内完成，但是第二种情形需要$O(n^3)$的时间，因此算法的总时间为$O(n^3)$。到目前为止，还没有出现具有更短时间的算法。然而，可有效地计算最小体积箱体的一种近似（Barequet 和 Har-Peled 1999）。

基于一个箱面包含一个多面体面的箱体的计算利用了旋转测径法。有$O(n)$个多面体面需要处理，每一个都需要$O(n)$时间，因此总时间为$O(n^2)$，与在上一段中所提及的相同。给定一个多面体的面，多面体在这个面的平面上的投影产生一个凸多边形。如果$\vec{n}$是这个面的指向外的法线向量，那么投影到多边形上的多面体的边和面就是法线$\vec{m}$满足$\vec{n}\cdot\vec{m}>0$的多边形和法线满足$\vec{n}\cdot\vec{m}<0$的多边形之间的分隔。该箱体在这个平面上的投影是一个矩形。旋转测径法用来寻找包含这个凸多边形的最小面积矩形。这等价于寻找指定的多面体面的最小体积箱体。

可以用简单的方法来处理三条边支持箱体的情形，即迭代三条边的所有组合，总共有$O(n^3)$种可能。对于互相垂直的三条边的每一种组合，构建该方向上的最小体积箱体。能够减少所需时间的原因之一，是互相垂直的三条边的组合可在处理多面体的面时找到，所需时间为$O(n^2)$。

这个多面体的最小体积箱体就是由面和由边组合所构建的所有箱体中的最小值。

13.11.3 最小面积的圆

一种寻找特定的有界圆的$O(n)$级方法是计算包含点集的最小面积的轴对齐矩形，然后选择一个外接于该矩形的圆。在大多数情形中，这个圆并不是包含点集的最小面积的圆。实际上，输入的点有时严格地位于这个圆内。例如，对于点集$\{(\pm 2,0),(0,\pm 1)\}$，就会出现这种情形。这个有界圆的圆心位于（0，0），其半径为$\sqrt{5}$。从原点到输入点的最大距离是2。许多的应用程序要求更合适的圆。

一个有界圆的支持点是刚好位于圆上的一个输入点。包含点集的最小面积的圆显然必须由至少两个输入点所支持；否则，目标圆可被缩小直到它与另一个输入点接触为止。即使点集可以具有多于两个的输入点，最小面积的圆也很可能只有两个输入点。例如，点集$\{(-1,0),(0,0),(1,0)\}$是共线的。包含它们的最小面积的圆的圆心为（0，0），半径为 1。其支持点为（±1，0）。在其他的例子中，支持点的数量为三。刚好位于最小面积的圆上的输入点的数量可能为四或者更多。但是三个就足够了，因为三个不共线的点惟一地确定一个

圆（参见 13.10 节）。

由于至少要有两个输入点位于这个圆上，因此一种吸引人的想法是假设这两个点必定为相隔最远的两个点。基于如下的反例可知，实际上并非如此。设输入点为$\{(1,0), (-1/2, \sqrt{3}/2), (-1/2, -\sqrt{3}/2), (-3/4, 0)\}$。这些点构成一个凸四边形。最前面的三个点构成一个等边三角形，其边长为$\sqrt{3}$。从$(1,0)$到$(-3/4,0)$的距离为$7/4 > \sqrt{3}$。因此，$(1,0)$和$(-3/4,0)$构成输入点中相隔最远的点对。最小面积有界圆为包含等边三角形，圆心位于$(0,0)$，半径为 1 的圆。包含$(1,0)$，$(-1/2, \sqrt{3}/2)$和$(-3/4,0)$的圆的圆心为$(1/8, \sqrt{3}/8)$，半径为$\sqrt{13}/4 < 1$，但是$(-1/2, -\sqrt{3}/2)$并不在该圆内。包含正相反的两个点$(-3/4,0)$和$(1,0)$的圆的圆心为$(1/8,0)$，半径为$7/8$，但是$(-1/2, \pm\sqrt{3}/2)$并不在这个圆内，因为从这些点到圆心的距离约为$1.068 > 7/8$。

用一种进行彻底搜索的方法可以得到答案，但是速度很慢。分析点集内的所有三元组。包含三个点的最小面积的圆是外接圆或三个点中有两个是相对的点的圆。特别地，当三个点共线时，就会出现这种情形。在这一过程中，在分析时记录最小半径的有界圆。最后，我们得到输入点集的最小面积的圆。这种算法是$O(n^3)$时间级的。

更有效的方法是，使一个圆增大，以包含所有的点。最初的圆是包含最前面的两个输入点的圆。检测其他的每一个点是否位于这个圆内。如果所有的点都包含在内，其中有一些可能位于圆上，那么最初的圆就是最小面积的圆。但是，我们一般并不会这样幸运，很难出现这种情形。更可能的情形是，剩余的点中的一个点Q位于最初的圆外，如果出现这种情形，最初的圆并未足够大，必须使其增大以包含点Q。实际上，点Q将用来作为新圆的支持点。其中的一个问题是，在遇到点Q之前已经进行了许多点在圆内的检测（如图 13.80 所示）。当最初的圆被修改为新的圆时，在最初的圆内的点可能并不位于被修改的圆内。算法一般必须重新开始，必须检测所有的点是否都位于新的圆内。

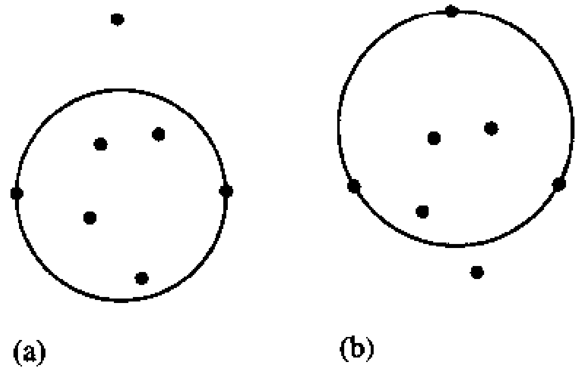


图 13.80　（a）当前的有界圆和圆外的一个点，该点将使圆增大。（b）新的有界圆，但原来在旧圆内的一个点现在位于新圆的外面，它将使算法重新开始执行

如果最初的圆是最小面积的圆，那么通过检测$n-2 = O(n)$个点就能确定。如果只需要重新开始m次，并且与n相比，m是一个很小的常数，那么算法是$O(n)$级的。然而，如果m是一个与n相当的数，那么算法的表现将比$O(n)$级更差。为了说明这一点，考虑位于半圆上的点，$P_i = (\cos\theta_i, \sin\theta_i)$，其中$\theta_i = \pi i/(n-1)$，$0 \leqslant i < n$。最初的有界圆由$P_0$和$P_1$所支持。下一个点$P_2$位于这个圆外，因此算法需要重新开始。新的圆由$P_0$和$P_2$所支持。点$P_1$位

于这个圆内，但是P_3位于这个圆外。在第i次迭代中，当前的有界圆由P_0和P_i所支持，点P_j，其中$0 < j < i$，位于这个圆内，但是P_{i+1}不位于圆内。即这个算法必须每次都重新开始。第i次迭代要求i次点在圆内的检测。只有当搜索到点P_{n-1}，并且实际上所有点都位于圆内时，我们才知道这是最小面积的圆。点在圆内的总的检测次数为$\sum_{i=1}^{n-1} = n(n-1)/2 = O(n^2)$。这种类型的更加复杂的例子甚至可能导致$O(n^3)$ 级的表现，与彻底搜索方法相同。

更仔细地分析上述的半圆例子，假设我们不按前面给出的次序来处理点，随机地改变这些点的次序，然后处理。为了便于讨论，设P_0为一个支持点。如果在改变序列的集合中的第二个点是P_j，其中j接近于$n-1$，那么最初的圆是非常大的。在有序的情形中，得到这个圆将需要进行j次迭代。在改变序列的情形中，我们节省了许多的时间。另一个在改变序列的情形中处理的点P_k，会使算法重新开始，很可能具有大于$j+1$的索引k。我们希望重新开始的次数m将是一个与n相比很小的常数，在这种情形中，算法是$O(n)$级的。

这种方法的形式化描述可在 Welzl（1991）中找到，它是一类叫做随机线性算法的其中一种。输入数据的改变序列预期的时间为$O(n)$ 级。这并不表示所有输入数据集都将运行这样的时间级。输入点的序列改变可能（虽然不是非常可能）导致一种引起超级线性表现的次序。例如，半圆问题的序列改变可能被证明是一致的，在这种情形中，这个例子需在$O(n^2)$ 级时间内完成。假设序列改变的分布是均衡的，在这个例子中，序列改变是一致的可能性是$1/n!$，对于大数n来说，这是一个很小的数。当然，其他的仅引起较小变化的序列改变将得到类似的较慢的圆构建，但是正如所说明的，期望的时间为$O(n)$。这种概念适用于更高的维，在下面将讨论三维空间问题。在d维空间中，期望的点在圆内的检测次数维$n(d+1)$。也就是说，渐近常数近似于$(d+1)!$。

该算法的递归公式为

```
Circle MinimumAreaCircle(PointSet Input, PointSet Support)
{
    if (Input is not empty) {
        P = GetRandomElementOf(Input);
        Input' = Input - {P};  // remove P from Input
        C = MinimumAreaCircle(Input', Support);
        if (P is inside C) {
            return C;
        } else {
            Support' = Support + {P}';  // add P to Support
            return MinimumAreaCircle(Input', Support');
        }
    } else {
        return CircleOf(Support);
    }
}
```

非递归公式为

```
Circle MinimumAreaCircle(int N, Point P[N])
{
    randomly permute P[0] through P[N - 1];
    C = ExactCircle1(P[0]);  // center P[0], radius 0
```

```
        Support = { P[0] };

        i = 1;
        while (i < N) {
            if (P[i] is not an element of Support) {
                if (P[i] is not contained by C)  {
                    C = Update(P[i], Support);
                    i = 0;  // restart the algorithm for the new circle
                    continue;
                }
            }
            i++;
        }

        return C;
    }
```

函数Update需要做的一项工作是将P[i]添加到支持点集Support中，并删除Support的其他因为添加P而不再是支持点的元素。请注意：当使用浮点数算术时，必须小心地实现这个函数。其中的问题是，将检测老的支持点是否包含在支持点和P[i]的不同组合内。这些组合中的一个在理论上必须包含所有的支持点。然而，数值舍入误差可能导致一种情形，即似乎没有任何一种组合能够包含所有的点。即使在进行点在圆内的检测时引入数值容差，还是可能遇到问题。一种解决方法是，当似乎没有圆能包含所有的支持点时，捕获这种情形，并使用与位于这个圆外的不合要求的点最接近的圆，与其他的圆的不合要求的点进行比较。对不同组合的圆的构建依赖于是否存在计算两个点或者三个点的最小面积的圆的函数。当然，这种算法的一种实现必须提供这类函数。

另一个可能与浮点相关的问题是更新调用函数总是将一个新圆赋予当前的最小面积的圆C。当使用浮点算术时，可能遇到一种情形，即由于交替交换的支持点集中两个点的循环，而导致循环成为无限的。在理论上，这个问题为更新所返回的圆的半径比当前的圆C的半径更大。然而，数值舍入误差将导致返回的圆的半径变得更小，因而引起无限循环。其解决方法是用如下的代码段替换包含更新函数的代码段：

```
    Circle tmp = Update(P[i], Support);
    if (tmp.radius > C.radius) {
        C = tmp;
        i = 0;
        continue;
    }
```

实现中需要考虑的其他问题是必须处理包括重复的输入点和在数值上几乎相同的不同的输入点的问题。包含三个并不共线的点的圆的构建说明了一种实现方法——使用线性系统方法来寻找圆心和半径的求解三角形的外接圆问题的方法。如果三个点几乎共线，那么这个系统的行列式可能接近于零，因此这种实现方法应该适当地处理这种情形，即检测出这种情形，并使用计算包含两个点的最小面积的圆的方法（从几乎共线的点中排除正确的点）。

13.11.4 最小体积的球

可用用一种类似于在二维空间中寻找包含点集的最小面积的圆的方法，来处理构建包含输入点集的最小体积的球的问题。我们建议阅读前一节关于圆的问题的内容，以理解这种方法的直观表述和思想。

在初级图形学程序员中普遍存在着的一种误解，就是认为通过将球心选择为输入点集的平均位置，然后确定距离球心最远的输入点以得到半径，就能构建包含输入点的最小的球。虽然这是一种可行而且易于实现的计算一个有界球的算法，但是求得的球却并不一定就是具有最小体积的球。

Welzl（1991）中的随机算法可应用于一般维的情形，特别是三维情形。其中递归公式与二维情形的递归公式完全一样，但是该函数计算的是球而不是圆。

```
Sphere MinimumVolumeSphere(PointSet Input, PointSet Support)
{
    if (Input is not empty) {
        P = GetRandomElementOf(Input);
        Input' = Input - {P};  // remove P from Input
        S = MinimumVolumeSphere(Input', Support);
        if (P is inside S) {
            return S;
        } else {
            Support' = Support + {P}';  // add P to Support
            return MinimumVolumeSphere(Input', Support');
        }
    } else {
        return SphereOf(Support);
    }
}
```

其中的非递归公式也与二维情形中的非递归公式类似：

```
Sphere MinimumVolumeSphere(int N, Point P[N])
{
    randomly permute P[0] through P[N - 1];
    S = ExactCircle1(P[0]);  // center P[0], radius 0
    Support = { P[0] };

    i = 1;
    while (i < N) {
        if (P[i] is not an element of Support) {
            if (P[i] is not contained by S) {
                S = Update(P[i], Support);
                i = 0;  // restart the algorithm for the new sphere
                continue;
            }
        }
        i++;
    }
```

```
        return S;
    }
```

在三维实现中必须处理在二维情形中出现的数值问题。参见 13.11.3 节的最后一段，其中讨论了需要考虑的问题，以及如何在实现中处理它们。

13.11.5 杂项

有时应用程序要使用其他类型的最小面积或者最小体积的有界区域。一般地，无论从开发算法和实现算法的角度来看，构建这类有界区域都是非常具有挑战性的。

1. 最小面积的椭圆

作为一个例子，考虑寻找包含点集的最小面积的椭圆的问题。可以直接扩展寻找最小面积的圆的算法（Gaertner 和 Schoenherr 1998）。在圆的问题中，支持点集的更新要求构建支持点集中二个或者三个点的最小面积的圆。这类问题被称为小问题，它们的解用于确定原来的具有 n 个点的大问题。有界椭圆的小问题涉及计算三个、四个或者五个点的最小面积的椭圆。对于三个不共线的点 $P_i, 0 \leqslant i \leqslant 2$，包含这些点的最小面积的椭圆的方程为 $(X-C)^{\mathrm{T}}\mathrm{M}(X-C)=2$，其中

$$C=\frac{1}{3}\sum_{i=0}^{2}P_i$$

即这些点的平均值，且 M 为 2×2 的矩阵，其逆为

$$\mathrm{M}^{-1}=\frac{1}{3}\sum_{i=0}^{2}(P_i-C)(P_i-C)^{\mathrm{T}}$$

对于构成一个凸多边形的五个点，最小面积的椭圆就是与这五个点匹配的椭圆。表示椭圆、双曲线或者抛物线的一般二次方程为 $x^2+axy+by^2+cx+dy+e=0$。从这个一般二次方程可以建立五个线性方程——即一个易于求解的系统，就能求得方程的五个系数。

计算包含构成一个凸四边形的四个点的最小面积的椭圆更困难一些。为了说明这个问题的复杂性，考虑一种特殊情形，即当点为 $(0,0)$, $(1,0)$, $(0,1)$ 和 (u,v) 时的情形，其中 $u>0, v>0$，并且 $u+v>1$。以这四个点为解的二次方程为 $x^2+bxy+cy^2-x-cy=0$，其中 $c>0$, $b^2<4c$，并且 $b=(1-u)/v+c(1-v)/u$。其中 c 是独立变量，因此存在无数包含这四个点的椭圆。要求解的问题是构建最小面积的椭圆。关于 c 的函数的面积可表示为

$$\mathrm{Area}(c)=\frac{\pi c(1-b+c)}{(4c-b^2)^{3/2}}$$

当 c 使面积函数对 c 的导数为零时，即 $\mathrm{Area}'(c)=0$ 时，面积为最小值。这样可得到一个关于 c 的三次多项式方程

$$P(c;u,v)=S(v)c^3+T(u,v)c^2-T(v,u)c-S(u)=0$$

其中 $S(v)=v^3(v-1)^2$ 且 $T(u,v)=uv^2(2v^2+uv+u-3v+1)$。$P$ 的最大根提供了正确的 c 值。在 $c=1$ 处 $P(1;u,v)=0$ 时，得到面积的最小值。当 $u=v$ 或者 $u^2+uv+v^2-u-v=0$（或者 $u+v=1$ 或 $u=-v$）时将出现这种情形。这些曲线将有效的 (u,v) 区域分解成为 $c>1$ 或者 $c<1$ 子区域。在数值上，通过对 $P(c)=0$ 应用牛顿方法并设初始估值为 $c=1$，可以在 $c<1$

的区域找到最大的根。在$c>1$的区域，牛顿方法可以应用于逆转的多项式方程$P(1/c)=0$，并设初始估值为$1/c=1$。

2. 具有固定中心和方向的最小面积的椭圆

求最小面积的椭圆问题的一种特殊情形是选择一个中心和一个方向，然后计算具有这个中心和方向的最小面积的椭圆的椭圆轴的长度。由于输入点可以用以指定的中心为原点和指定的方向为坐标轴的坐标系的左边来表示，因此我们可以在中心为原点并且方向为单位矩阵的条件下分析这个问题。这时的椭圆方程为$(x/a)^2+(y/b)^2=1$，并且椭圆的面积为πab，我们要针对输入点集求其最小值。

对轴长的限制条件为$a>0$且$b>0$。额外的限制条件包括要求每一个点(x_i,y_i)都位于椭圆内，$(x_i/a)^2+(y_i/b)^2\leqslant 1$。要求解的问题就是在满足所有的不等式条件时使二次函数ab最小化。设$u=1/a^2$和$v=1/b^2$。等价的问题是，在满足线性不等式限制条件$u\geqslant 0$，$v\geqslant 0$和$x_i^2u+y_i^2v\leqslant 1$对所有i成立时使$f(u,v)=uv$最大化。这是一个二次方程的编程问题，因此可以使用求解这类问题的一般方法（Pierre 1986）。这类编程也出现在其他的计算几何应用程序中，许多研究者都探讨过这类问题（例如，Gaertner 和 Schoenherr 2000）。

虽然在此可以利用一般二次方程的编程方法，但是可以使用更加几何化的方法来求解这类问题。$f(u,v)$的定义域由一个凸多边形所包围，这个凸多边形具有边$u=0$，$v=0$，以及其他在第一象限内由点在椭圆内的包含性限制条件所确定的边。并非所有的限制条件都在定义域内。f的最大值必须出现在凸多边形的边界上（不包括$u=0$和$v=0$），因此可用一种巧妙的多边形搜索来得到最大化的(u,v)。这个点可以是一个顶点或者边的一个内点。可用一种线性搜索来确定在u轴上产生最小v值的限制直线。分析其他的限制直线，求得它们与这条最初的直线的交点，以找到最近的交点$(0,v)$。这种搜索可以得到这个凸多边形的第一条边。如果f在内点或者u为最小值的端点处取最大值，那么问题已被解决。否则，f在边上的最大值出现于u为最大值的端点。相对于得到u的最大值的限制直线，来重复处理对限制直线排序。在迭代时，随着限制直线被处理，和/或被确定不属于这个凸多边形的边界，对它们做标记，以避免重复处理它们。

3. 最小体积的椭球

计算包含一个三维空间的点集的最小体积的椭球的算法也类似于计算最小体积的球的算法。球的小问题与寻找包含两个、三个或者四个点的最小球相关。对于一个椭球，小问题涉及 4~9 个点。对于构成一个凸多边形的 9 个点，求得一个以一般二次方程的 9 个未知系数为未知数的 9 个线性方程的解，就能计算出椭球。对于构成一个凸多边形的四个点，有一个代数公式用于计算最小体积的椭球。其中心为

$$C=\frac{1}{4}\sum_{i=0}^{3}P_i$$

即这些点的平均值，以及M为 3×3 的矩阵，其逆为

$$\mathrm{M}^{-1}=\frac{1}{4}\sum_{i=0}^{3}(P_i-C)(P_i-C)^{\mathrm{T}}$$

对于 $5 \leqslant n \leqslant 8$ 的情形，要计算最小体积椭圆非常困难。体积函数取决于 $9-n$ 个变量（二次方程的系数）。计算关于这些变量的 $9-n$ 个导数，并将它们设为零，每一个方程都可简化为一个多项式方程。当 $n=8$ 时，要求解的方程是一个具有一个未知数的多项式方程，这是一个易于处理的问题。然而，当 $n=5$ 时，要求解的方程包括四个具有四个未知数的多项式方程。可用消元理论（Wee 和 Goldman 1995a，1995b）来减少变量，但是这样做会出现许多数值问题，并且得到的含有一个变量的多项式方程将具有非常高的次数，因此求根就要遇到很多的数值问题。另一种方法是用数值方法来求解多项式方程系统。现在还不知道有没有人能建立最小体积的椭球算法的一个健壮实现。

4．求最小值的数值方法

虽然在计算几何学中可能没有吸引力，但是实际上可以使用数值方法递归地求解最小面积或者最小体积的有界问题。在圆、球、椭圆或者椭球的情形中，这些二次对象的方程具有未知的系数，这些系数应满足基于点在对象内所要求的不等式限制条件。导出以这些未知系数为变量的面积和体积的公式。结果为一个函数，要求其满足一组不等式约束条件的最小值，这是一种非线性的编程方法。在工业应用中，这种方法的吸引人之处在于，问题很容易构建起来，并能利用现有的健壮的非线性程序库来求解这类问题。为了缩短开发时间，必须牺牲纯几何方法的速度和精度，这在计算机科学中是一种可行的折中，但是学术界的研究人员一般并不考虑这种方法。

13.12 面积和体积测量

本节将描述计算多边形面积的算法，包括二维空间和三维空间的多边形的面积，以及计算多面体体积的算法。分别用代数、几何和分析的方法来说明不同的算法。

13.12.1 二维多边形的面积

考虑一个三角形 (V_0, V_1, V_2)，它的顶点次序为逆时针次序。设 $V_i=(x_i, y_i)$，可以利用基础代数和三角几何方法来证明三角形的面积为

$$\mathrm{Area}(V_0, V_1, V_2)=\frac{1}{2}\det\begin{bmatrix}1 & 1 & 1\\ x_0 & x_1 & x_2\\ y_0 & y_1 & y_2\end{bmatrix} \tag{13.1}$$

显然，$\mathrm{Area}(V_1, V_2, V_0)=\mathrm{Area}(V_2, V_0, V_1)=\mathrm{Area}(V_0, V_1, V_2)$，因为这与从哪个顶点开始逆时针排序无关。然而，如果次序是顺时针方向的，那么 $\mathrm{Area}(V_0, V_2, V_1)=\mathrm{Area}(V_2, V_1, V_0)=\mathrm{Area}(V_1, V_0, V_2)=-\mathrm{Area}(V_0, V_1, V_2)$，全部都是负数。因此，对于三个顶点 U, V 和 W 的任何集合，方程（13.1）所定义的函数 $\mathrm{Area}(U, V, W)$ 叫做由顶点所构成的三角形的有符号面积。如果顶点是逆时针方向排序的，有符号面积为正。如果顶点是顺时针方向排序的，有符号面积为负。如果顶点是共线的，则有符号面积为零。

1．作为代数量的面积

设 $V=(x, y)$ 为平面上的任意点。下面的代数恒等式成立：

$$\text{Area}(V_0, V_1, V_2) = \text{Area}(V, V_0, V_1) + \text{Area}(V, V_1, V_2) + \text{Area}(V, V_2, V_0) \tag{13.2}$$

可用如下的方法来检验该恒等式：展开方程右边的行列式并进行一些代数运算，以证明得到的结果与左边的行列式完全相同。

可以推导出简单多边形$\mathcal{P}$的面积公式，方程（13.2）的几何直观可以帮助这种推导。逆时针次序的顶点V_0到V_{n-1}及任意一个点V所确定的面积为

$$\begin{aligned}\text{Area}(\mathcal{P}) = {} & \text{Area}(V, V_0, V_1) + \text{Area}(V, V_1, V_2) + \cdots + \text{Area}(V, V_{n-2}, V_{n-1}) \\ & + \text{Area}(V, V_{n-1}, V_0)\end{aligned} \tag{13.3}$$

利用数学归纳法可以证明这个公式。假设这个公式对所有具有n个顶点的简单多边形都成立。现在考虑一个具有$n+1$个顶点的多边形$\mathcal{P}'$。正如在 13.9 节中所提到的，一个多边形必须至少有一个耳，一个三角形包含除构成三角形的顶点之外的其他任何多边形的顶点。重新标记$\mathcal{P}'$的顶点，使得它的耳$\mathcal{T}$为三角形(V_{n-1}, V_n, V_0)，而且从$\mathcal{P}'$中删除这个耳所得到的多边形$\mathcal{P}$为$\langle V_0, \ldots, V_{n-1}\rangle$。$\mathcal{T}$的面积为

$$\text{Area}(\mathcal{T}) = \text{Area}(V, V_{n-1}, V_n) + \text{Area}(V, V_n, V_0) + \text{Area}(V, V_0, V_{n-1})$$

这可由方程 13.2 得出。$\mathcal{P}$的面积为

$$\begin{aligned}\text{Area}(\mathcal{P}) = {} & \text{Area}(V, V_0, V_1) + \text{Area}(V, V_1, V_2) + \cdots + \text{Area}(V, V_{n-2}, V_{n-1}) \\ & + \text{Area}(V, V_{n-1}, V_0)\end{aligned}$$

这可根据假设推导得出。$\mathcal{P}'$的面积是组合的面积，即$\text{Area}(\mathcal{P}') = \text{Area}(\mathcal{T}) + \text{Area}(\mathcal{P})$。当这两个表达式相加在一起时，$\text{Area}(\mathcal{T})$的项$\text{Area}(V, V_0, V_{n-1})$与$\text{Area}(\mathcal{P})$的项$\text{Area}(V, V_{n-1}, V_0)$相抵消。最后的和为

$$\begin{aligned}\text{Area}(\mathcal{P}') = {} & \text{Area}(V, V_0, V_1) + \text{Area}(V, V_1, V_2) + \cdots + \text{Area}(V, V_{n-1}, V_n) \\ & + \text{Area}(V, V_n, V_0)\end{aligned}$$

因此这个公式对任何具有$n+1$个顶点的多边形$\mathcal{P}'$成立。根据数学归纳法原理，这个公式对所有$n \geqslant 3$的整数都成立。

对每一个项$\text{Area}(\vec{0}, V_i, V_{i+1})$运用$V = (0,0)$, $V_i = (x_i, y_i)$和方程（13.1），方程（13.3）右边的表达式将导出一个可用计算机程序来实现的公式

$$\text{Area}(\mathcal{P}) = \frac{1}{2}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) \tag{13.4}$$

其中的索引需与n进行模运算。即$x_n = x_0$和$y_n = y_0$。方程（13.4）的和的每一项都要求两次乘法和一次减法。进行简单的移项可将其减少为每一项仅需一次乘法和一次减法：

$$\begin{aligned}\sum_{i=0}^{n-1}(x_i y_{i+1} - x_{i+1} y_i) &= \sum_{i=0}^{n-1}[x_i(y_{i+1} - y_{i-1}) + x_i y_{i-1} - x_{i+1} y_i] \\ &= \sum_{i=0}^{n-1} x_i(y_{i+1} - y_{i-1}) + \sum_{i=0}^{n-1}(x_i y_{i-1} - x_{i+1} y_i)\end{aligned}$$

$$= \sum_{i=0}^{n-1} x_i(y_{i+1} - y_{i-1}) + x_0 y_{-1} - x_n y_{n-1}$$

$$= \sum_{i=0}^{n-1} x_i(y_{i+1} - y_{i-1})$$

基于索引需与n进行模运算的假设，有$x_n = x_0$和$y_{-1} = y_{n-1}$，因此最后的等式成立。利用一个简单的代数事实，可以更有效地计算面积：

$$\text{Area}(\mathcal{P}) = \frac{1}{2}\sum_{i=0}^{n-1} x_i(y_{i+1} - y_{i-1}) \tag{13.5}$$

网络新闻组 *comp.graphics.algorithms* 的 FAQ 说，这个公式是在 Dan Sunday（2001）中首先提出的，但是这个公式早就已经出现在网络新闻组中，Dave Rusin（1995）将其张贴到网络新闻组 *sci.math* 上。由于其简单性，可能 1995 年之前就已经有人发现了这个公式。

如果多边形$\mathcal{P}$不是简单多边形，而是包含一个本身也是一个多边形的洞，那么面积将是外面的多边形所包围的面积与里面的多边形所包围的面积之差。如果你的多边形数据结构允许将这样的多边形存储在一个数组中，使得里面的多边形总是位于每一条边的左边，那么里面的多边形将是顺时针次序的。在这种情形中，方程（13.5）依然成立，并不需要分别处理外面的多边形和里面的多边形。

2．作为几何量的面积

方程（13.2）具有几何上的等价证明。图 13.81 显示了一个三角形和V的两个候选点。由V和对V可见的一条边所构成的三角形的有符号面积为负。其他三角形的有符号面积为正。注意，不论如何选择V，三角形的重叠部分的有符号面积都相互抵消，而仅留下原始三角形的面积。

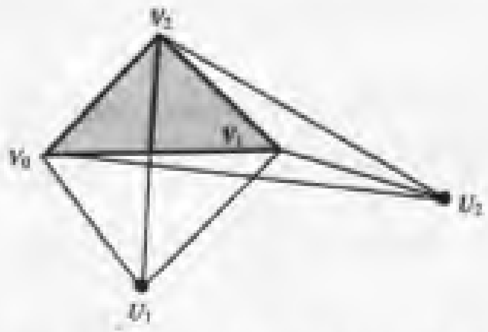


图 13.81　选择点 U_1 和 U_2 计算方程 13.2。三角形只有一条边对第一个点可见。三角形的两条边对第二个点可见

3．作为分析量的面积

二维空间中的简单多边形$\mathcal{P}$的面积也可用微积分学中的分析方法导出。设多边形的顶点为$V_i = (x_i, y_i)$，其中$0 \leqslant i \leqslant n-1$，它们是逆时针排序的。这个多边形的内部区域就是一个互连的区域，用R表示，因此$\mathcal{P}$是区域R的边界。将格林定理应用于一个边界为$\mathcal{P}$的简单互连区域R，就可得到一个计算这个多边形所封闭的面积的公式：

$$\iint_R \vec{\nabla} \cdot F \, dx \, dy = \oint_{\mathcal{P}} F \cdot \vec{n} \, ds$$

其中 $F(x, y) = (F_1(x, y), F_2(x, y))$ 是一个可微向量域，$\vec{\nabla} \cdot F = \partial F_1/\partial x + \partial F_2/\partial y$ 是这个向量域的散度，而 $\vec{n}$ 是 $\mathcal{P}$ 的指向外的法线向量。可以在许多正式的教科书中找到这个公式（例如，Finney 和 Thomas 1996）。如果 $\mathcal{P}$ 是用 $(x(t), y(t))$ 参数表示的，其中 $t \in [a, b]$，则一条切线向量为 $\vec{t} = (x'(t), y'(t))$，其中的导数是关于 t 的导数，并且指向外的法线为 $\vec{n} = (y'(t), -x'(t))$。积分公式变为

$$\iint_R \vec{\nabla} \cdot F \, dx \, dy = \int_a^b F(x(t), y(t)) \cdot (y'(t), -x'(t)) \, dt$$

当选择 $F = (x, y)/2$ 时，就得到格林定理的面积公式。此时，$\vec{\nabla} \cdot F \equiv 1$ 并且左边的积分表示 R 的面积，右边的积分所确定的值，即在 R 的边界 $\mathcal{P}$ 上的积分为：

$$\text{Area}(R) = \frac{1}{2} \int_a^b x(t)y'(t) - y(t)x'(t) \, dt \tag{13.6}$$

这个多边形的每一条边都可参数化表示为 $(x_i(t), y_i(t)) = V_i + t(V_{i+1} - V_i)$，其中 $t \in [0, 1]$。在 t 上的积分成为

$$\begin{aligned}
\int_a^b x(t)y'(t) - y(t)x'(t) \, dt &= \sum_{i=0}^{n-1} \int_0^1 x_i(t)y_i'(t) - y_i(t)x_i'(t) \, dt \\
&= \sum_{i=0}^{n-1} \int_0^1 [x_i + t(x_{i+1} - x_i)][y_{i+1} - y_i] \\
&\quad - [y_i + t(y_{i+1} - y_i)][x_{i+1} - x_i] \, dt \\
&= \sum_{i=0}^{n-1} [x_i + (x_{i+1} - x_i)/2][y_{i+1} - y_i] \\
&\quad - [y_i + (y_{i+1} - y_i)/2][x_{i+1} - x_i] \, dt \\
&= \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)
\end{aligned}$$

其中 $x_n = x_0$ 和 $y_n = y_0$。这其实就是方程 13.4，当时是用代数方法推出的。

13.12.2 三维多边形的面积

考虑一个三维空间的多边形 $\mathcal{P}$，它在平面 $\hat{n} \cdot X = c$ 上封闭一个简单的互连区域，其中 $\hat{n}$ 是单位长度向量。设顶点为 V_i，其中 $0 \leqslant i \leqslant n - 1$，并假设当你从 $\hat{n}$ 所指向的一边观察这个多边形时，其顶点是逆时针排序的。将方程（13.5）应用于三维空间多边形相对于平面的坐标。如果 $\hat{u}$ 和 $\hat{w}$ 是单位长度向量，并且 $\hat{u}, \hat{w}$ 和 $\hat{n}$ 是互相垂直的，那么 $V_i = x_i\hat{u} + y_i\hat{w} + c\hat{n}$，并且对于所有的 i，平面点为 $(x_i, y_i) = (\hat{u} \cdot V_i, \hat{w} \cdot V_i)$。面积为

$$\text{Area}(\mathcal{P}) = \frac{1}{2} \sum_{i=0}^{n-1} (\hat{u} \cdot V_i)(\hat{w} \cdot (V_{i+1} - V_{i-1})) \tag{13.7}$$

这个公式虽然在数学上成立，但需要更多的时间来进行评测。

1、投影得到的面积

更有效的一种方法是，注意到如果$\hat{n}=(n_0,n_1,n_2)$是一个单位长度的法线向量，并且$n_2\neq 0$，那么法线为$\hat{n}$，位于平面上的对象的面积为

$$\text{Area(object)} = \text{Area}(\text{Projection}_{xy}(\text{object}))/|n_2|$$

也就是说，计算$\mathcal{P}$在xy平面上的投影的面积，并用这条法线的z分量来校验。结果是函数图形的表面积公式的一个简单的推导。如果平面为$\hat{n}\cdot(x,y,z)=c$并且$n_2\neq 0$，那么$z=f(x,y)=(c-n_0x-n_1y)/n_2$。关于区域$R$上的$(x,y)$的函数$f$的图形的表面积为

$$\iint_R \sqrt{1+f_x^2+f_y^2}\,dx\,dy$$

其中$f_x=\partial f/\partial x$并且$f_y=\partial f/\partial y$是$f$的一阶偏导数。在前面提到的平面的情形中，$f_x=-n_0/n_2, f_y=-n_1/n_2$，并且

$$\begin{aligned}\sqrt{1+f_x^2+f_y^2} &= \sqrt{1+\left(\frac{n_0}{n_2}\right)^2+\left(\frac{n_1}{n_2}\right)^2}\\ &= \sqrt{\frac{n_0^2+n_1^2+n_2^2}{n_2^2}}\\ &= \frac{1}{|n_2|}\end{aligned}$$

其中这个分数的分子为 1，因为假设$\hat{n}$是单位长度的。因此，$\iint_R \sqrt{1+f_x^2+f_y^2}\,dx\,dy = \iint_R 1/|n_2|dxdy = \text{Area}(R)/|n_2|$。

因此，通过计算仅使用多边形顶点的(x,y)分量得到的二维空间多边形的面积，然后除以平面法线的z分量的绝对值，就能得到法线为$\hat{n}$的位于平面上的多边形的面积。然而，如果n_2接近于零，很可能引起数值问题。最好根据$|n_2|$, $|n_1|$或$|n_0|$中哪一个最大来决定使用(x,y)，(x,z)或者(y,z)。最终的公式为

$$\text{Area}(\mathcal{P}) = \frac{1}{2}\begin{cases}\frac{1}{|n_2|}\sum_{i=0}^{n-1} x_i(y_{i+1}-y_{i-1}), & |n_2|=\max_i |n_i|\\ \frac{1}{|n_1|}\sum_{i=0}^{n-1} x_i(z_{i+1}-z_{i-1}), & |n_1|=\max_i |n_i|\\ \frac{1}{|n_0|}\sum_{i=0}^{n-1} y_i(z_{i+1}-z_{i-1}), & |n_0|=\max_i |n_i|\end{cases} \tag{13.8}$$

2. 由斯托克斯定理得到的面积

另一个计算面积的公式出现在 Arvo（1991）中的一篇标题为“平面多边形的面积和多面体的体积”的文章中。这个公式为

$$\text{Area}(R) = \frac{1}{2}\hat{n}\cdot\sum_{i=0}^{n-1}(P_i\times P_{i+1}) \tag{13.9}$$

给定逆时针次序，出现在实际公式中的绝对值是不需要的。如果具有顶点的有序集，这将是很有用的，但是你并不知道它是顺时针的还是逆时针的。替换三元常数积中的公式

$P_i = x_iU + y_iV + c\hat{n}$，可得$\hat{n} \cdot P_i \times P_{i+1} = x_iy_{i+1} - x_{i+1}y_i$，这就说明这两个公式是等价的。然而，可以将$\hat{n}$作为因子提到和式之外，以将 n 次点积减少为一次点积。每一项$P_i \times P_{i+1}$都要求 6 次乘法和 3 次加法。n 次叉积的和要求 3（n–1）次加法。与$\hat{n}$的点积要求 3 次乘法和两次加法，并且最后与二分之一相乘要求一次乘法。总的计算要求 $6n$+4 次乘法和 $6n$–1 次加法。显然，计算方程（13.8）的效率比计算方程（13.9）的效率更高。

在 Arvo（1991）的文章中提到，可以根据斯托克斯定理导出方程（13.9），斯托克斯定理为

$$\iint_S \vec{\nabla} \times F \cdot \hat{n}\, d\sigma = \oint_C F \cdot d\vec{R}$$

其中 S 是边界为有界曲线C的流式曲面。向量域 F 是 S 在每一个位置上的法线。$F = F_1(x, y, z), F_2(x, y, z), F_3(x, y, z)$的旋度为

$$\vec{\nabla} \times F = \left(\frac{\partial F_3}{\partial y} - \frac{\partial F_2}{\partial z}, \frac{\partial F_1}{\partial z} - \frac{\partial F_3}{\partial x}, \frac{\partial F_2}{\partial x} - \frac{\partial F_1}{\partial y}\right)$$

微分$d\vec{R} = (dx, dy, dz)$仅限制于曲线C，微分$d\sigma$表示曲面S的无限小元素。

对于三维空间多边形的情形，S 是由多边形包围的平面区域，C就是多面体本身，$\hat{n}$是一个单位长度法线向量（相对于$\hat{n}$，多边形的顶点是按逆时针排序的），并且 $F = \hat{n} \times (x, y, z)/2$，此时，$\vec{\nabla} \times F = \hat{n}$。面积为

$$\begin{aligned}\text{Area}(S) &= \iint_S d\sigma = \iint_S \hat{n} \cdot \hat{n}\, d\sigma = \oint_C \frac{1}{2}\hat{n} \times (x, y, z) \cdot (dx, dy, dz)\\ &= \oint_C \frac{1}{2}\hat{n} \cdot (x, y, z) \times (dx, dy, dz)\end{aligned}$$

如果C是由$(x(t), y(t))$（$t \in [a, b]$）参数表示的，那么公式变成

$$\begin{aligned}\text{Area}(S) &= \frac{1}{2}\int_a^b \hat{n} \cdot (y(t)z'(t) - z(t)y'(t), z(t)x'(t) - x(t)z'(t), x(t)y'(t)\\ &\quad - y(t)x'(t))\, dt\\ &= \hat{n} \cdot \left(\frac{1}{2}\int_a^b y(t)z'(t) - z(t)y'(t)\, dt, \frac{1}{2}\int_a^b z(t)x'(t) - x(t)z'(t)\, dt,\right.\\ &\quad \left.\frac{1}{2}\int_a^b x(t)y'(t) - y(t)x'(t)\, dt\right)\end{aligned}$$

注意到每一个积分都是显示在方程（13.6）中的形式。因此，这个多边形的面积是将原多边形投影到三个坐标平面上得到的三个平面多边形面积的加权和，其中的权就是法线向量的分量。每一个积分都由方程（13.5）关于适当的坐标给出，因此

$$\begin{aligned}\text{Area}(S) &= \frac{1}{6}\hat{n} \cdot \left(\sum_{i=0}^{n-1} y_i(z_{i+1} - z_{i-1}), \sum_{i=0}^{n-1} z_i(x_{i+1} - x_{i-1}), \sum_{i=0}^{n-1} x_i(y_{i+1} - y_{i-1})\right)\\ &= \frac{1}{2}\hat{n} \cdot \sum_{i=0}^{n-1} P_i \times P_{i+1}\end{aligned}$$

这就是方程 13.9。

13.12.3 多面体的体积

本节的讨论是计算二维空间中多边形面积的思想的直接扩展。考虑一个多面体 (V_0, V_1, V_2, V_3)。设 $V_i = (x_i, y_i, z_i)$，可以利用基础代数和三角几何方法来证明多面体的有符号体积为

$$\text{Volume}(V_0, V_1, V_2, V_3) = \frac{1}{6} \det \begin{bmatrix} 1 & 1 & 1 & 1 \\ x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \end{bmatrix} \tag{13.10}$$

假设这些点不全共面，序列改变后的一半产生正值，另一半产生负值。

设 $V = (x, y, z)$ 为空间内的任意点。下面的代数恒等式成立：

$$\begin{aligned} \text{Volume}(V_0, V_1, V_2, V_3) &= \text{Volume}(V, V_0, V_1, V_2) + \text{Volume}(V, V_1, V_2, V_3) \\ &\quad + \text{Volume}(V, V_2, V_3, V_0) + \text{Volume}(V, V_3, V_0, V_1) \end{aligned} \tag{13.11}$$

可用如下的方法来检验该恒等式：展开方程右边的行列式并进行一些代数运算，以证明得到的结果与左边的行列式完全相同。其几何方法与二维空间中的情形完全相同。有的有符号体积为正，有的有符号体积为负。只要共用 V 的两个多面体重叠并且具有相反的有符号体积，那么它们将互相抵消。

将方程（13.3）扩展到三维空间，并利用扩展后的方程就能计算所有的面都是三角形的简单多面体的体积。当从多面体外观察时，假设多面体的面都是逆时针方向排序的。点 V 是任意的；在实际中将其选择为零向量 $\vec{0}$。

$$\text{Volume}(P) = \sum_{\text{face } F} \text{Volume}(\vec{0}, F.V_0, F.V_1, F.V_2) \tag{13.12}$$

下面列出了利用微积分方法的分析构建过程。考虑一个多面体，表示为 S，它封闭空间中的一个简单的互连区域 R。设多面体的 n 个面称为 S_i，其中 $0 \leqslant i < n$。设每一个面 S_i 都有指向外的单位长度法线 $\hat{n}_i$ 和顶点 $P_{i,j}$，其中 $0 \leqslant j < m(i)$，当从外面观察时，它们是逆时针排序的，这里强调，顶点的总数 $m(i)$ 取决于面 i。利用散度定理可以得出这个多面体所包围的体积公式，散度定理是将格林定理从二维空间扩展到三维空间的直接推广，即对于具有边界曲面 S 的一个简单互连区域 R，有

$$\iiint_R \vec{\nabla} \cdot F \, dx\, dy\, dz = \iint_S F \cdot \hat{n} \, d\sigma$$

其中 $F(x, y, z) = (F_1(x, y, z), F_2(x, y, z), F_3(x, y, z))$ 是一个微分向量域，$\vec{\nabla} \cdot F = \partial F_1/\partial x + \partial F_2/\partial y + \partial F_3/\partial z$ 是该向量域的散度，$\hat{n}$ 是曲面 S 指向外的法线向量，而 $d\sigma$ 表示这个曲面上的无限小元素。可以在许多正式教材中找到这个公式（例如，Finney 和 Thomas 1996）。

当选择 $F = (x, y, z)/3$ 时，就得到散度定理的体积公式。此时，$\vec{\nabla} \cdot F = 1$ 并且

$$\text{Volume}(R) = \iiint_R dx\, dy\, dz = \frac{1}{3} \iint_S \hat{n} \cdot (x, y, z) \, d\sigma$$

由于这个多面体是一个不相交的并集$S=\bigcup_{i=0}^{n-1}S_i$，右边的积分成为多个积分的和，每一个积分都与多面体的一个多边形面相关

$$\iint_S \hat{n}\cdot(x,y,z)\,d\sigma=\sum_{i=0}^{n-1}\iint_{S_i}\hat{n}_i\cdot(x,y,z)\,d\sigma$$

其中$\hat{n}_i$是S_i的指向外的法线。S_i的平面为$\hat{n}_i\cdot(x,y,z)=c_i$，$c_i$为某些常数。位于多边形上的任何点都确定该常数，特别地，$c=\hat{n}_i\cdot P_{0,i}$。这个积分进一步简化为

$$\sum_{i=0}^{n-1}\iint_{S_i}\hat{n}_i\cdot(x,y,z)\,d\sigma=\sum_{i=0}^{n-1}\iint_{S_i}c_i\,d\sigma=\sum_{i=0}^{n-1}c_i\ \text{Area}(S_i)$$

替换方程（13.9）中的公式，可得体积公式为

$$\text{Volume}(R)=\frac{1}{6}\sum_{i=0}^{n-1}\left((\hat{n}_i\cdot P_{0,i})\hat{n}_i\cdot\sum_{j=0}^{m(i)-1}(P_{i,j}\times P_{i,j+1})\right)\tag{13.13}$$

附录A 数 值 方 法

A.1 求解线性系统

$m \times n$线性方程系统的一般形式为

$$
\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,n}x_n &= c_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,n}x_n &= c_2 \\
&\vdots \\
a_{m,1}x_1 + a_{m,2}x_2 + \cdots + a_{m,n}x_n &= c_n
\end{aligned}
$$

线性系统经常写成矩阵形式：

$$\mathrm{A}\vec{x} = \vec{c}$$

$$
\begin{bmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\
a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\
 & & \vdots & \\
a_{m,1} & a_{m,2} & \cdots & a_{m,n}
\end{bmatrix}
\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}
=
\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}
$$

矩阵A表示系数矩阵（coefficient matrix），而矩阵

$$
\begin{bmatrix}
a_{1,1} & a_{1,2} & \cdots & a_{1,n} & c_1 \\
a_{2,1} & a_{2,2} & \cdots & a_{2,n} & c_2 \\
 & & \vdots & & \\
a_{m,1} & a_{m,2} & \cdots & a_{m,n} & c_n
\end{bmatrix}
$$

叫做增广矩阵。

A.1.1 特例：求解三角形系统

为了引入求解方形线性系统的一般方法，我们先分析一种特殊情形，即三角形系统。这样的系统满足$a_{i,j} = 0$（$i > j$）。这种系统之所以被称为三角形系统，是因为其特有的形状像三角形，即：

$$
\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \cdots\cdots\cdots\cdots\cdots\cdots + a_{1,n}x_n &= c_1 \\
a_{2,2}x_2 + \cdots\cdots\cdots\cdots\cdots\cdots + a_{2,n}x_n &= c_2 \\
&\vdots \\
a_{n-2,n-2}x_{n-2} + a_{n-2,n-1}x_{n-1} + a_{n-2,n}x_n &= c_{n-2}
\end{aligned}
$$

$$a_{n-1,n-1}x_{n-1}+a_{n-1,n}x_n=c_{n-1}$$
$$a_{n,n}x_n=c_n$$

我们可用一种叫做回代（back substitution）的技术来求解这种系统。如果我们注意其中的最后一个方程，就会发现它仅有一个未知数(x_n)，很容易求解：

$$x_n=\frac{c_n}{a_{n,n}}$$

我们因此求得了一个解（x_n），我们可以将它代入倒数第二个方程，又可以简单地求得x_{n-1}：

$$x_{n-1}=\frac{c_{n-1}-a_{n-1,n}\frac{c_n}{a_{n,n}}}{a_{n-1,n-1}}$$

依次类推进行回代，直到我们回到第一个方程，并求得第一个未知数x_1。一般地，通过将前一次计算得到的$x_n, x_{n-1}, \cdots, x_{k+1}$来求解第$k$个方程中的$x_k$：

$$x_k=\frac{c_k-\sum_{m=k+1}^{n}a_{k,m}x_m}{a_{k,k}}$$

A.1.2 高斯消元法

高斯消元法可能是最常用的求解一般线性系统的方法。这种方法的基础是我们前面介绍的回代技术：首先，将系统一步一步地变换成三角形式，然后应用前面介绍的回代方法。实际上，我们已经在2.4.2节中见过了使用这种方法的一个（简单的）例子。

高斯注意到一个线性方程系统经过一定的线性变换后仍然具有相同的解集。这类变换包括：

（1）交换两行的次序

（2）用一个非零因子乘以一行

（3）用两行之和来替代其中的任一行

在早先的例子中，我们已经看到过（2）和（3），但是你可能想知道交换两行的次序有什么用。在无限精确的数学世界中，这种方法是没什么用的，但是在用计算机来实现的世界中，这种变换是非常有用的。

考虑摘自Johson和Riess（1982）的一个由具有两个未知数的两个方程所组成的简单系统：

$$\begin{aligned}(1)\quad 0.0001x_1+x_2&=1\\(2)\quad x_1+x_2&=2\end{aligned}$$

如果我们想消除（2）中的x_1，就需要对（1）进行如下的乘法：

$$\begin{array}{rrcr}-10000\times(1): & -x_1-10000x_2 & = & -10000\\ 1\times(2): & x_1+x_2 & = & 2\\ \hline \text{Sum:} & -9999x_2 & = & -9998\end{array}$$

这样可得到一个新的系统：

$$\text{(1)}\quad 0.0001x_1 + x_2 = 1$$
$$\text{(2)}\quad -9999x_2 = -9998$$

如果我们通过如下的计算来完成求解过程

$$x_2 = \frac{9998}{9999} \approx 0.99989999$$

并将其代回（1）:

$$0.0001x_1 + x_2 = 1$$
$$0.0001x_1 + \frac{9998}{9999} = 1$$
$$0.0001x_1 = 1 - \frac{9998}{9999}$$
$$0.0001x_1 = \frac{1}{9999}$$
$$x_1 = \frac{10000}{9999}$$
$$x_1 \approx 1.00010001$$

这样做的一个“问题”是，上述计算的假设前提是精度是无限的。然而，考虑如果出现如下情形，会出现怎样的结果：我们的计算结果是六位数的，但是仅仅存储三位数。那么，如果我们还是用−10000来与（1）相乘，将得到

$$\begin{array}{rrcr} -10000 \times (1): & x_1 - 10000x_2 & = & -10000 \\ 1 \times (2): & x_1 + x_2 & = & 2 \\ \hline \text{Sum:} & -10000x_2 & = & -10000 \end{array}$$

由于舍入误差，可得$x_2 = 1$。如果我们将其回代到（1），可得

$$0.0001x_1 + x_2 = 1$$
$$0.0001x_1 + 1 = 1$$
$$0.0001x_1 = 0$$
$$x_1 = 0$$

但这样的结果完全是错误的。当然，这个例子是专门设计用于显示最坏的情形的，但是截去和舍入误差将趋向于使它们在后续的步骤中变得复杂化。

另一个与精度相关的问题可能出现于回代的除法运算中。回忆一下，基本的回代步骤是

$$x_k = \frac{c_k - \sum_{m=k+1}^{n} a_{k,m} x_m}{a_{k,k}}$$

即系数出现在分母中。由于计算机在进行除法运算时，当除数具有尽可能大的绝对值时，可以得到最精确的结果，因此，如果具有最大系数（与所有的$x_{k,n}$相比）的x_k被用于除法，就可以得到最佳的结果。

下面是一个例子，假设我们具有如下的具有三个未知数的系统：

$$(1) \quad x_1 - 3x_2 + 2x_3 = 6$$
$$(2) \quad 2x_1 - 4x_2 + 2x_3 = 18$$
$$(3) \quad -3x_1 + 8x_2 + 9x_3 = -9$$

现在设想如何用计算机程序来求解它，我们需要“重新组织”上述系统，使得x_1的最大系数首先出现

$$(1) \quad -3x_1 + 8x_2 + 9x_3 = -9$$
$$(2) \quad 2x_1 - 4x_2 + 2x_3 = 18$$
$$(3) \quad x_1 - 3x_2 - 2x_3 = 6$$

然后再用2乘以（1），用3乘以（2），并相加，得到一个新的（2），再用1乘以（1），用3乘以（3），得到一个新的（3）

$$\begin{array}{lrcr}
2 \times (1): & -6x_1 + 16x_2 + 18x_3 & = & -18 \\
3 \times (2): & 6x_1 - 12x_2 + 6x_3 & = & 54 \\
\hline
\text{Sum:} & 4x_2 + 24x_3 & = & 36 \\
1 \times (1): & -3x_1 + 8x_2 + 9x_3 & = & -9 \\
3 \times (3): & 3x_1 - 9x_2 - 6x_3 & = & 18 \\
\hline
\text{Sum:} & -1x_2 + 3x_3 & = & 9
\end{array}$$

这样就得到--个新的系统：

$$(1) \quad x_1 - 3x_2 + 2x_3 = 6$$
$$(2) \quad 4x_2 + 24x_3 = 36$$
$$(3) \quad -1x_2 + 3x_3 = 9$$

然后，我们可用4乘以（3），并与（2）相加，得到一个新的（3）

$$\begin{array}{lrcr}
1 \times (2) & 4x_2 + 24x_3 & = & 36 \\
4 \times (3) & -4x_2 + 12x_3 & = & 36 \\
\hline
\text{Sum:} & 36x_3 & = & 72
\end{array}$$

由此可以求得解为$(1, -3, -2)$。

上述基本算法的伪码为

```
// j indexes columns (pivots)
for (j = 1 to n - 1) do
    // Pivot step
    find l such that a_{l,j} has the largest value among (a_{j,j}, a_{j+1,j}, ..., a_{n,j})
    exchange rows l and j

    // Elimination step
    // k indexes rows
    for (k = 1 to n) do
        // Form multiplier
        m = -a_{k,j} / a_{j,j}
        // Multiply row j by m and add to row k
        for (i = j + 1 to n) do
            a_{k,i} = a_{k,i} + m a_{k,j}
```

```
        // Multiply and add constant for row j
        c_k = c_k + m c_j

// Back substitute
x_n = c_n / a_{n,n}
for (k = n - 1 downto 1) do
    x_k = (c_k - Σ_{m=k+1}^{n} a_{k,m} x_m) / a_{k,k}
```

A.2 多项式系统

上一节说明了如何求解线性方程系统。给定 n 个具有 m 个未知数的方程，$\sum_{j=0}^{m} a_{ij}x_j = b_i$（$0 \leqslant i < n$），该系统的矩阵表示形式为$\mathbf{A}\vec{x} = \vec{b}$，其中$\mathbf{A} = [a_{ij}]$是$n \times m$，$\vec{x} = [x_j]$是$m \times 1$的，$\vec{b} = [b_i]$是$n \times 1$的。建立起$n \times (m+1)$的增广矩阵$[\mathbf{A}|\vec{b}]$，并减少其行数，得到$[\mathbf{E}|\vec{c}]$，这个矩阵具有如下的性质：

- 每一行的第一个非零项都是 1。
- 如果第 r 行中第一个非零项是在第 c 列中，那么第 c 列中的其他项都是 0。
- 所有为零的行都出现在矩阵的最后。
- 如果第 1 行到第 r 行中第一个非零项出现在第c_1到第c_r列中，那么$c_1 < \cdots < c_r$。

如果存在一个前 m 项都是零但是最后一项是非零的行，那么该方程系统无解。如果不存在这样的行，用$\rho = \text{rank}([\mathbf{E}|\vec{c}])$表示该增广矩阵中非零行的数量。如果$\rho = m$，则系统有且仅有一个解。此时，$\mathbf{E} = \mathbf{I}_m$是$m \times m$的单位矩阵，并且其解为$\vec{x} = \vec{c}$。如果$\rho < m$，则系统具有无穷个解，解集的维数为$m - \rho$。在这种情形中，可以省略为零的行，得到$\rho \times (m+1)$的矩阵$[\mathbf{I}_\rho|\mathbf{F}|\vec{c}_+]$，其中$\mathbf{I}_\rho$是$\rho \times \rho$的单位矩阵，$\mathbf{F}$ 是$\rho \times (m-\rho)$的，并且$\vec{c}_+$由 $\vec{c}$ 的前面ρ项组成。设$\vec{x}$被分解为前面ρ个分量$\vec{x}_+$和后面$m-\rho$个分量$\vec{x}_-$。该系统的一般解为$\vec{x}_+ = \vec{c}_+ - \mathbf{F}\vec{x}_-$，其中$\vec{x}_-$为系统的自由参数。

求解方形系统（$n=m$）的基础数值线性系统的求解算法使用了消行的方法，使得（1）算法所需的时间级很小，在这种情形中为$O(n^3)$；（2）在出现浮点值的系统中计算也是健壮的。利用余因子扩展来求解线性系统也是可能的，但是这样的算法的时间级为 $O(n!)$，对于较大的 n 来说，这种算法是很费时的。然而，对于许多计算机图形应用来说，$n=3$。上述的消行基础算法一般需要比简单余因子扩展算法需要更多的时间周期，而且实际应用的系数矩阵一般不是退化（或近似退化）矩阵，因此健壮性并不是一个问题，所以，对于这样大小的系统来说，余因子扩展算法是一种更好的选择。

在计算机图形应用程序中，多项式方程系统也会产生退化问题。例如，在二维空间中确定两个圆的交点等价于求解两个具有两个未知数的二次方程。确定三维空间中两个椭球是否相交，等价于确定一个由三个具有三个未知数的二次方程所组成的系统是否具有实数解。计算一条直线与一个多项式面的交点需要建立并求解多项式方程系统。求解这类系统的一种方法与用于求解线性系统的消元方法几乎一样。然而，前者除了消行法之外，还具有另一种方法，即余因子扩展法。

A.2.1 一个形式变量的线性方程

为了说明一般的求解思想，考虑具有变量x的一个方程$a_0+a_1x=0$。如果$a_1\neq 0$，存在惟一解$x=-a_0/a_1$。如果$a_1=0$且$a_0\neq 0$，则没有解。如果$a_0=a_1=0$，则任何x都是一个解。

现在考虑具有相同变量的两个方程，$a_0+a_1x=0$和$b_0+b_1x=0$，其中$a_1\neq 0$且$b_1\neq 0$。用b_1乘以第一个方程，用a_1乘以第二个方程，再将两个方程相减，可得$a_0b_1-a_1b_0=0$。这是一个值x同时为两个方程的解的必需条件。如果该条件满足，那么求解第一个方程，可得$x=-a_0/a_1$。根据前一节中介绍的求解线性系统的消行方法，$n=2$，$m=1$，并且增广矩阵的消去步骤为

$$\begin{bmatrix} a_1 & -a_0 \\ b_1 & -b_0 \end{bmatrix} \sim \begin{bmatrix} a_1b_1 & -a_0b_1 \\ a_1b_1 & -a_1b_0 \end{bmatrix} \sim \begin{bmatrix} a_1b_1 & -a_0b_1 \\ 0 & a_0b_1-a_1b_0 \end{bmatrix} \sim \begin{bmatrix} 1 & -a_0/a_1 \\ 0 & a_0b_1-a_1b_0 \end{bmatrix}$$

条件$a_0b_1-a_1b_0=0$其实就是前一节中提到的保证至少存在一个解的条件。

这里展示的消行法是一种规范的方法。解的存在性和解x本身都可通过系统的参数a_0，a_1，b_0和b_1的函数来求得。这些参数并不要求是已知的常数，它们本身也可依赖于其他变量。假设$a_0=c_0+c_1y$和$b_0=d_0+d_1y$。原来的两个方程为$a_1x+c_1y+c_0=0$和$b_1x+d_1y+d_0=0$，这是一个具有两个未知数的两个方程所组成的系统。存在解的条件为$0=a_0b_1-a_1b_0=(c_0+c_1y)b_1-b_0(d_0+d_1y)=(b_1c_0-b_0d_0)+(b_1c_1-b_0d_1)y$。开始是具有未知数$x$和$y$的两个方程，消去$x$，得到一个关于$y$的方程，最后得到的结果就是这个条件。只要$b_1c_1-b_0d_1\neq 0$，关于$y$的方程就具有唯一的一个解。一旦计算出$y$，就可以计算出$a_0=c_0+c_1y$和$x=-a_0/a_1$。

我们再次修改问题，再额外设定$a_1=e_0+e_1y$和$b_1=f_0+f_1y$。这两个方程为

$$\begin{aligned} e_1xy+e_0x+c_1y+c_0&=0 \\ f_1xy+f_0x+d_1y+d_0&=0 \end{aligned}$$

这是一个由两个具有两个未知数的二次方程构成的系统。它存在解的条件为

$$\begin{aligned} 0&=a_0b_1-a_1b_0 \\ &=(c_0+c_1y)(f_0+f_1y)-(e_0+e_1y)(d_0+d_1y) \\ &=(c_0f_0-e_0d_0)+((c_0f_1-e_0d_0)+(c_1f_0-e_1d_0))y+(c_0f_1-e_1d_1)y^2 \end{aligned}$$

该方程的y至多具有两个实数解。每一个解都可得到一个$x=-a_0/a_1=-(c_0+c_1y)/(e_0+e_1y)$的值。这两个方程定义了平面上的双曲线，它们的渐近线与轴平行。从几何意义上来说，这两条双曲线至多只能相交于两个点。

当存在另外的线性方程时，可以采用相似的方法。例如，如果$a_0+a_1x=0, b_0+b_1x=0$和$c_0+c_1x=0$，成对地求解它们可得到存在解的条件为：$a_0b_1-a_1b_0=0$和$a_0c_1-a_1c_0=0$。如果两个条件都满足，那么一个解为$x=-a_0/a_1$。如果允许$a_0=a_{00}+a_{10}y+a_{01}z$，$b_0=b_{00}+b_{10}y+b_{01}z$和$c_0=c_{00}+c_{10}y+c_{01}z$，则可得到三个具有三个未知数的方程。解存在的两个条件为两个关于y和z的线性方程，消去了变量x。通过用同样的方法消去y，可以进一步对这两个方程进行消元。注意，在使用这种方法时，存在许多种的$AB-CD$的形式。这

些项本质上就是增广矩阵的 2×2 子矩阵的行列式。

A.2.2 一个形式变量的任意次方方程

考虑关于 x 的多项式方程 $f(x)=\sum_{i=0}^{n}a_i x^i=0$。当 $n\leqslant 4$ 时，可以通过近似解法来求解方程的根；而对于任意次方的方程，可以通过数值解法来求方程的根。如果还有第二个关于同样变量的多项式方程 $g(x)=\sum_{j=0}^{m}b_j x^j=0$，要解决的问题就是确定存在一个解的条件，就像我们在上一节中所做的那样。其前提条件是 $a_n\neq 0$ 且 $b_m\neq 0$。上一节处理了 $n=m=1$ 的情形。

1. $n=2$ 且 $m=1$ 的情形

这种情形中，方程的形式为 $f(x)=a_2x^2+a_1x+a_0=0$ 和 $g(x)=b_1x+b_0=0$，其中 $a_2\neq 0$ 且 $b_1\neq 0$。还必须满足下式

$$0=b_1f(x)-a_2xg(x)=(a_1b_1-a_2b_0)x+a_0b_1=:c_1x+c_0 \tag{A.1}$$

其中系数 c_0 和 c_1 由上述最后一个等式确定。上述两个方程现在简化为两个线性方程 $b_1x+b_0=0$ 和 $c_1x+c_0=0$。

与上一节的情形相比，现在我们需要多做一些工作。在本节中，我们设定如下的前提，即方程的第一个系数是非零的（$b_1\neq 0$ 且 $c_1\neq 0$）。在当前的处理中，c_1 是从上述的指定信息中导出的，因此我们需要处理当它为零时的情形。如果 $c_1=0$，那么 $c_0=0$ 是存在一个解的必要条件。由于根据假设前提有 $b_1\neq 0$，$c_0=0$ 隐含了 $a_0=0$。条件 $c_1=0$ 隐含了 $a_1b_1=a_2b_0$。当 $a_0=0$ 时，二次方程的一个解为 $x=0$。要使它同时也是 $g(x)=0$ 的一个解，必须有 $0=g(0)=b_0$，其中又隐含了 $0=a_2b_0=a_1b_1$，或者由于 $b_1\neq 0$，其实就是 $a_1=0$。简单地说，这就是 $f(x)=a_2x^2$ 和 $g(x)=b_1x$ 的情形。同时，当 $a_0=0$ 时，二次方程的另一个根可由 $a_2x+a_1=0$ 确定。该方程和 $b_1x+b_0=0$ 就是上一节所讨论的情形，可进行适当的简化。

我们也可以直接求解 $x=-b_0/b_1$，将其代入二次方程，乘以 b_1^2，以获得存在解的条件 $a_2b_0^2-a_1b_0b_1+a_0b_1^2=0$。

2. $n=2$ 且 $m=2$ 的情形

这种情形中，方程为 $f(x)=a_2x^2+a_1x+a_0=0$ 和 $g(x)=b_2x^2+b_1x+b_0=0$，其中 $a_2\neq 0$ 且 $b_2\neq 0$。还必须满足如下的条件

$$0=b_2f(x)-a_2g(x)=(a_1b_2-a_2b_1)x+(a_0b_2-a_2b_0)=:c_1x+c_0 \tag{A.2}$$

这两个二次方程被简化为一个线性方程，其系数 c_0 和 c_1 由上述的等式定义。如果 $c_1=0$，那么存在解的条件还必须满足 $c_0=0$。在这种情形中，考虑

$$0=b_0f(x)-a_0g(x)=(a_2b_0-a_0b_2)x^2+(a_1b_0-a_0b_1)x$$

$$=-c_0x^2+(a_1b_0-a_0b_1)x=(a_1b_0-a_0b_1)x$$

如果 $a_1b_0-a_0b_1\neq 0$，那么方程的解必定为 $x=0$，其结果为 $0=f(0)=a_0$ 和 $0=g(0)=b_0$。但是这与 $a_1b_0-a_0b_1\neq 0$ 矛盾。因此，如果 $a_1b_2-a_2b_1=0$ 且 $a_0b_2-a_2b_0=0$，那么必定有 $a_1b_0-a_0b_1=0$。这三个条件隐含了 $(a_0,a_1,a_2)\times(b_0,b_1,b_2)=(0,0,0)$，因此 (b_0,b_1,b_2) 是 (a_0,a_1,a_2) 的乘积，并且这两个二次方程实际上就是同一个方程。既然如此，如

果$c_1 \neq 0$，我们就能将问题简化为$n=2$且$m=1$的情形。前面已经讨论过这种情形。

另一种方法是计算$a_2 g(x) - b_2 f(x) = (a_2 b_1 - a_1 b_2)x + (a_2 b_0 - a_0 b_2) = 0$和$b_1 f(x) - a_1 g(x) = (a_2 b_1 - a_1 b_2)x^2 + (a_0 b_1 - a_1 b_0) = 0$。求解第一个方程中的$x$，$x = (a_0 b_2 - a_2 b_0)/(a_2 b_1 - a_1 b_2)$，再将其代入第二个方程，用其中的分母乘以第二个方程，可得

$$(a_2 b_1 - a_1 b_2)(a_1 b_0 - a_0 b_1) - (a_2 b_0 - a_0 b_2)^2 = 0$$

3. $n \geqslant m$的一般情形

消元处理是递归的。由于我们已经建立了次数小于n的情形下的消元过程，因此只需将当前情形中次数为n的$f(x)$和次数为$m \leqslant n$的$g(x)$简化为减小次数的方程。这里假设$a_n \neq 0$且$b_m \neq 0$。

定义$h(x) = b_m f(x) - a_n x^{n-m} g(x)$。条件$f(x)=0$和$g(x)=0$隐含了

$$\begin{aligned}
0 &= h(x) \\
&= b_m f(x) - a_n x^{n-m} g(x) \\
&= b_m \sum_{i=0}^{n} a_i x^i - a_n x^{n-m} \sum_{i=0}^{m} b_i x^i \\
&= \sum_{i=0}^{n} a_i b_m x^i - \sum_{i=0}^{m} a_n b_i x^{n-m+i} \\
&= \sum_{i=0}^{n-m-1} a_i b_m x^i + \sum_{i=n-m}^{n-1} (a_i b_m - a_n b_{i-(n-m)}) x^i
\end{aligned}$$

其中$\sum_{i=0}^{-1}(*) = 0$（当上标索引小于下标索引时，其和为零）。多项式$h(x)$的次数至少为$n-1$。

这样，多项式$g(x)$和$h(x)$的次数都小于n，所以可以利用处理较小次数的算法来求解它们。

A.2.3 任意个形式变量的任意次方方程

一般的多项式方程系统总是可以表示为关于其中一个变量的多项式方程系统的形式。正如上一节所讨论的，存在解的条件就是关于其余变量的新的多项式方程。而且，这些方程一般比原来的方程具有更高的次数。由于进行了消元，简化所得的方程的次数增加了。系统最终简化为一个关于一个变量的（高次）多项式方程。如果求得了该方程的解，可以将它们代入上述的存在解的条件中，以求解其他的解。这与求解线性系统的回代法相似。

1. 两个变量，一个二次方程，一个线性方程

方程为$Q(x,y) = \alpha_{00} + \alpha_{10}x + \alpha_{01}y + \alpha_{20}x^2 + \alpha_{11}xy + \alpha_{02}y^2 = 0$和$L(x,y) = \beta_{00} + \beta_{10}x + \beta_{01}y = 0$。它们可被表示为关于$x$的多项式的形式，即

$$f(x) = (\alpha_{20})x^2 + (\alpha_{11}y + \alpha_{10})x + (\alpha_{02}y^2 + \alpha_{01}y + \alpha_{00}) = a_2 x^2 + a_1 x + a_0$$

和

$$g(x) = (\beta_{10})x + (\beta_{01}y + \beta_{00}) = b_1 x + b_0$$

$f(x)=0$和$g(x)=0$存在解的条件为$h(x) = h_0 + h_1 x + h_2 x^2 = 0$，其中

$$h_0 = \alpha_{02}\beta_{00}^2 - \alpha_{01}\beta_{00}\beta_{01} + \alpha_{00}\beta_{01}^2$$

$$h_1 = \alpha_{10}\beta_{01}^2 + 2\alpha_{02}\beta_{00}\beta_{10} - \alpha_{11}\beta_{00}\beta_{01} - \alpha_{01}\beta_{01}\beta_{10}$$

$$h_2 = \alpha_{20}\beta_{01}^2 - \alpha_{11}\beta_{01}\beta_{10} + \alpha_{02}\beta_{10}^2$$

给定$h(x)=0$的一个根x，y的形式值可从$L(x,y)=0$中获得，即$y=-(\beta_{00}+\beta_{10}x)/\beta_{01}$。

【实例】 设$Q(x,y)=x^2+xy+y^2-1$和$L(x,y)=y-x+1$。求解$L(x,y)=0$可得$y=x-1$。将其代入$Q(x,y)=0$，得到$3x^2-3x=0$。其根为$x_0=0$和$x_1=1$。对应的y值为$y_0=x_0-1=-1$和$y_1=x_1-1=0$。点$(0,-1)$和$(1,0)$是$Q(x,y)=0$所定义的二次曲线与$L(x,y)=0$所定义的直线的交点。■

2. 两个变量，两个二次方程

考虑两个方程$F(x,y)=\alpha_{00}+\alpha_{10}x+\alpha_{01}y+\alpha_{20}x^2+\alpha_{11}xy+\alpha_{02}y^2=0$和$G(x,y)=\beta_{00}+\beta_{10}x+\beta_{01}y+\beta_{20}x^2+\beta_{11}xy+\beta_{02}y^2=0$。它们可被表示为关于$x$的多项式的形式，即

$$f(x)=(\alpha_{20})x^2+(\alpha_{11}y+\alpha_{10})x+(\alpha_{02}y^2+\alpha_{01}y+\alpha_{00})=a_2x^2+a_1x+a_0$$

和

$$g(x)=(\beta_{20})x^2+(\beta_{11}y+\beta_{10})x+(\beta_{02}y^2+\beta_{01}y+\beta_{00})=b_2x^2+b_1x+b_0$$

存在解的条件为

$$0=(a_2b_1-a_1b_2)(a_1b_0-a_0b_1)-(a_2b_0-a_0b_2)^2=\sum_{i=0}^{4}h_iy^i=:h(y)$$

其中

$$h_0=d_{00}d_{10}-d_{20}^2$$

$$h_1=d_{01}d_{10}+d_{00}d_{11}-2d_{20}d_{21}$$

$$h_2=d_{01}d_{11}+d_{00}d_{12}-d_{21}^2-2d_{20}d_{22}$$

$$h_3=d_{01}d_{12}+d_{00}d_{13}-2d_{21}d_{22}$$

$$h_4=d_{01}d_{13}-d_{22}^2$$

而且

$$d_{00}=\alpha_{22}\beta_{10}-\beta_{22}\alpha_{10}$$

$$d_{01}=\alpha_{22}\beta_{11}-\beta_{22}\alpha_{11}$$

$$d_{10}=\alpha_{10}\beta_{00}-\beta_{10}\alpha_{00}$$

$$d_{11}=\alpha_{11}\beta_{00}+\alpha_{10}\beta_{01}-\beta_{11}\alpha_{00}-\beta_{10}\alpha_{01}$$

$$d_{12}=\alpha_{11}\beta_{01}+\alpha_{10}\beta_{02}-\beta_{11}\alpha_{01}-\beta_{10}\alpha_{02}$$

$$d_{13}=\alpha_{11}\beta_{02}-\beta_{11}\alpha_{02}$$

$$d_{20}=\alpha_{22}\beta_{00}-\beta_{22}\alpha_{00}$$

$$d_{21}=\alpha_{22}\beta_{01}-\beta_{22}\alpha_{01}$$

$$d_{22}=\alpha_{22}\beta_{02}-\beta_{22}\alpha_{02}$$

计算$h(y)=0$的根。每一个根$\bar{y}$都可用于计算$f(x)=F(x,\bar{y})=0$和$g(x)=G(x,\bar{y})=0$。关于一个变量的两个二次方程具有一个解，该解由方程 A.2，$x=(a_2b_0-a_0b_2)/(a_1b_2-a_2b_1)$，所定义，其中$a_2=\alpha_{20}, a_1=\alpha_{11}\bar{y}+\alpha_{10}, a_0=\alpha_{02}\bar{y}^2+\alpha_{01}\bar{y}+\alpha_{00}, b_2=\beta_{20}, b_1=\beta_{11}\bar{y}+\beta_{10}$，而且$b_0=\beta_{02}\bar{y}^2+\beta_{01}\bar{y}+\beta_{00}$。

【实例】设$F(x,y)=x^2+y^2-1$和$G(x,y)=(x+y)^2+4(x-y)^2-4$。那么$h(y)=-36y^4+36y^2-1$，它的根为±0.169 102 和±0.985 599。对应的x值由$x=(a_2b_0-a_0b_2)/(a_1b_2-a_2b_1)=1/(6y)$，±0.169 102 和±0.985 599 所定义。■

3. 三个变量，一个二次方程，两个线性方程

设三个方程为$F(x,y,z)=\sum_{0\le i+j+k\le 2}\alpha_{ijk}x^iy^jz^k$，$G(x,y,z)=\sum_{0\le i+j+k\le 1}\beta_{ijk}x^iy^jz^k$和$H(x,y,z)=\sum_{0\le i+j+k\le 1}\gamma_{ijk}x^iy^jz^k$。作为关于$x$的多项式方程，它们可被表示为$f(x)=a_2x^2+a_1x+a_0=0, g(x)=b_1x+b_0=0$和$h(x)=c_1x+c_0=0$，其中

$$a_0=\sum_{0\le j+k\le 2}\alpha_{0jk}y^jz^k$$

$$a_1=\sum_{0\le j+k\le 1}\alpha_{1jk}y^jz^k$$

$$a_2=\alpha_{200}$$

$$b_0=\beta_{010}y+\beta_{001}z+\beta_{000}$$

$$b_1=\beta_{100}$$

$$c_0=\gamma_{010}y+\gamma_{001}z+\gamma_{000}$$

$$c_1=\gamma_{100}$$

$f=0$和$g=0$存在x解的条件为

$$0=a_2b_0^2-a_1b_0b_1+a_0b_1^2=\sum_{0\le i+j\le 2}d_{ij}y^iz^j=:D(y,z)$$

其中

$$d_{20}=\alpha_{200}\beta_{010}^2-\beta_{100}\alpha_{110}\beta_{010}+\beta_{100}^2\alpha_{020}$$

$$d_{11}=2\alpha_{200}\beta_{010}\beta_{001}-\beta_{100}(\alpha_{110}\beta_{001}+\alpha_{101}\beta_{010})+\beta_{100}^2\alpha_{011}$$

$$d_{02}=\alpha_{200}\beta_{001}^2-\beta_{100}\alpha_{101}\beta_{001}+\beta_{100}^2\alpha_{002}$$

$$d_{10}=2\alpha_{200}\beta_{010}\beta_{000}-\beta_{100}(\alpha_{110}\beta_{000}+\alpha_{100}\beta_{010})+\beta_{100}^2\alpha_{010}$$

$$d_{01}=2\alpha_{200}\beta_{001}\beta_{000}-\beta_{100}(\alpha_{101}\beta_{000}+\alpha_{100}\beta_{001})+\beta_{100}^2\alpha_{001}$$

$$d_{00}=\alpha_{200}\beta_{000}^2-\beta_{100}\alpha_{100}\beta_{000}+\beta_{100}^2\alpha_{000}$$

$g=0$和$h=0$存在x解的条件为

$$0=b_0c_1-b_1c_0=e_{10}y+e_{01}z+e_{00}=:E(y,z)$$

其中

$$e_{10}=\beta_{010}\gamma_{100}-\gamma_{010}\beta_{100}$$

$$e_{01}=\beta_{001}\gamma_{100}-\gamma_{001}\beta_{100}$$

$$e_{00}=\beta_{000}\gamma_{100}-\gamma_{000}\beta_{100}$$

我们现在有两个关于两个未知数的方程，一个二次方程$D(y,z)=0$和一个线性方程$E(y,z)=0$。这种情形已在上一节中得以处理。对于每一个解$(\bar{y},\bar{z})$，从方程（A.1），$x=a_0b_1/(a_2b_0-a_1b_1)$中可以计算一个对应的x值。

【实例】设$F(x,y,z)=x^2+y^2+z^2-1, G(x,y,z)=x+y+z$和$H(x,y,z)=x+y-z$。系数多项式为$a_2=1,a_1=0,a_0=y^2+z^2-1,b_1=1,b_0=y+z$，$c_1=1$和$c_0=y-z$。中间多项式为$D(y,z)=2y^2+2yz+2z^2-1$和$E(y,z)=2z$。条件$E(y,z)=0$隐含了$\bar{z}=0$。将其代入其他的中间多项式中，可得$0=D(y,0)=2y^2-1$，因此$\bar{y}=\pm1/\sqrt{2}$。对应的$x$值由$x=a_0b_1/(a_2b_0-a_1b_1)=(y^2+z^2-1)/(y+z)$确定，因此$\bar{x}=\mp1/\sqrt{2}$。存在两个交点，即$(-1/\sqrt{2},1/\sqrt{2},0)$和$(1/\sqrt{2},-1/\sqrt{2},0)$。■

4. 三个变量，两个二次方程，一个线性方程

设三个方程为$F(x,y,z)=\sum_{0\le i+j+k\le 2}\alpha_{ijk}x^iy^jz^k$，$G(x,y,z)=\sum_{0\le i+j+k\le 2}\beta_{ijk}x^iy^jz^k$和$H(x,y,z)=\sum_{0\le i+j+k\le 1}\gamma_{ijk}x^iy^jz^k$。作为关于$x$的多项式方程，它们可被表示为$f(x)=a_2x^2+a_1x+a_0=0, g(x)=b_2x^2+b_1x+b_0=0$和$h(x)=c_1x+c_0=0$，其中

$$a_0=\sum_{0\le j+k\le 2}\alpha_{0jk}y^jz^k$$

$$a_1=\sum_{0\le j+k\le 1}\alpha_{1jk}y^jz^k$$

$$a_2=\alpha_{200}$$

$$b_0=\sum_{0\le j+k\le 2}\beta_{0jk}y^jz^k$$

$$b_1=\sum_{0\le j+k\le 1}\beta_{1jk}y^jz^k$$

$$b_2=\beta_{200}$$

$$c_0=\gamma_{010}y+\gamma_{001}z+\gamma_{000}$$

$$c_1=\gamma_{100}$$

$f=0$和$h=0$存在x解的条件为

$$0=a_2c_0^2-a_1c_0c_1+a_0c_1^2=\sum_{0\le i+j\le 2}d_{ij}y^iz^j=:D(y,z)$$

其中

$$d_{20}=\alpha_{200}\gamma_{010}^2-\gamma_{100}\alpha_{110}\gamma_{010}+\gamma_{100}^2\alpha_{020}$$

$$d_{11}=2\alpha_{200}\gamma_{010}\gamma_{001}-\gamma_{100}(\alpha_{110}\gamma_{001}+\alpha_{101}\gamma_{010})+\gamma_{100}^2\alpha_{011}$$

$$d_{02}=\alpha_{200}\gamma_{001}^2-\gamma_{100}\alpha_{101}\gamma_{001}+\gamma_{100}^2\alpha_{002}$$

$$d_{10}=2\alpha_{200}\gamma_{010}\gamma_{000}-\gamma_{100}(\alpha_{110}\gamma_{000}+\alpha_{100}\gamma_{010})+\gamma_{100}^{2}\alpha_{010}$$

$$d_{01}=2\alpha_{200}\gamma_{001}\gamma_{000}-\gamma_{100}(\alpha_{101}\gamma_{000}+\alpha_{100}\gamma_{001})+\gamma_{100}^{2}\alpha_{001}$$

$$d_{00}=\alpha_{200}\gamma_{000}^{2}-\gamma_{100}\alpha_{100}\gamma_{000}+\gamma_{100}^{2}\alpha_{000}$$

$g=0$和$h=0$存在x解的条件为

$$0=b_2c_0^2-b_1c_0c_1+b_0c_1^2=\sum_{0\leq i+j\leq 2}e_{ij}y^iz^j=:E(y,z)$$

其中

$$e_{20}=\beta_{200}\gamma_{010}^{2}-\gamma_{100}\beta_{110}\gamma_{010}+\gamma_{100}^{2}\beta_{020}$$

$$e_{11}=2\beta_{200}\gamma_{010}\gamma_{001}-\gamma_{100}(\beta_{110}\gamma_{001}+\beta_{101}\gamma_{010})+\gamma_{100}^{2}\beta_{011}$$

$$e_{02}=\beta_{200}\gamma_{001}^{2}-\gamma_{100}\beta_{101}\gamma_{001}+\gamma_{100}^{2}\beta_{002}$$

$$e_{10}=2\beta_{200}\gamma_{010}\gamma_{000}-\gamma_{100}(\beta_{110}\gamma_{000}+\beta_{100}\gamma_{010})+\gamma_{100}^{2}\beta_{010}$$

$$e_{01}=2\beta_{200}\gamma_{001}\gamma_{000}-\gamma_{100}(\beta_{101}\gamma_{000}+\beta_{100}\gamma_{001})+\gamma_{100}^{2}\beta_{001}$$

$$e_{00}=\beta_{200}\gamma_{000}^{2}-\gamma_{100}\beta_{100}\gamma_{000}+\gamma_{100}^{2}\beta_{000}$$

我们现在有两个关于两个未知量的方程，二次方程$D(y,z)=0$和$E(y,z)=0$。这种情形已在前面的章节中得以处理。对于每一个解$(\bar{y},\bar{z})$，从方程（A.1），$x=a_0b_1/(a_2b_0-a_1b_1)$中可以计算一个对应的x值。线性方程可用来计算x，假设x的系数不为零。

【实例】 设$F(x,y,z)=x^2+y^2+z^2-1$（一个球），$G(x,y,z)=4x^2+9y^2+36z^2-36$（一个椭球）和$H(x,y,z)=x+y+z$（一个平面）。系数多项式为$a_2=1, a_1=0, a_0=y^2+z^2-1, b_2=4, b_1=0, b_0=9y^2+36z^2-36, c_1=1$和$c_0=y+z$。中间多项式为$D(y,z)=2y^2+2yz+2z^2-1$和$E(y,z)=13y^2+8yz+40z^2-36$。现在我们又得到了两个关于两个未知量的二次方程，前面已经解决了这类问题。通过消去y得到的二次多项式为$h(z)=-3556z^4+7012z^2-3481$。这个多项式没有实数解，因此该多项式系统没有实数解。■

【实例】 设$F(x,y,z)=x^2+y^2+z^2-1, G(x,y,z)=x^2+16y^2+36z^2-4$和$H(x,y,z)=x+y+8z$。系数多项式为$a_2=1,a_1=0,a_0=y^2+z^2-1,b_2=1$， $b_1=0,b_0=16y^2+36z^2-4$，$c_1=1$和$c_0=y+8z$。中间多项式为$D(y,z)=2y^2+16yz+65z^2-1$和$E(y,z)=17y^2+16yz+100z^2-4$。两个二次方程$D(y,z)=0$和$E(y,z)=0$简化为一个二次方程$0=h(z)=-953425z^4+27810z^2-81$。根$\bar{z}$为$\pm0.160893$和$\pm0.0572877$。$\bar{y}$值可用方程（A.2）来确定，注意，当前的问题是用$y$和$z$来表示的，而不是用$x$和$y$来表示的。该方程为$\bar{y}=(-905\bar{z}^2+9)/(240\bar{z})$，因此，对应于$\bar{z}$值的$\bar{y}$值为$\mp0.373\,626$和$\pm0.438\,568$。$\bar{x}$值由方程(A.1)，$\bar{x}=(\bar{y}^2+\bar{z}^2-1)/(\bar{y}+8\bar{z})$来确定。由数对$(\bar{y},\bar{z})=(-0.373\,627, 0.160\,893)$可得$\bar{x}=-0.913520$，因此交点为$(\bar{x},\bar{y},\bar{z})=(-0.913\,520,-0.373\,627,0.160\,893)$。其他的交点为$(0.913\,520,0.373\,627,-0.160\,893)$，$(-0.896\,87,0.438\,568,0.057\,287\,7)$，以及$(0.896\,87,-0.438\,568,-0.057\,287\,7)$。正如在讨论一般情形时所提及的，我们可以利用线性方程来求解$\bar{x}=-(\bar{y}+8\bar{z})$。■

5. 三个变量，三个二次方程

设三个方程为$F(x,y,z)=\sum_{0\leqslant i+j+k\leqslant 2}\alpha_{ijk}x^iy^jz^k$，$G(x,y,z)=\sum_{0\leqslant i+j+k\leqslant 2}\beta_{ijk}x^iy^jz^k$和$H(x,y,z)=\sum_{0\leqslant i+j+k\leqslant 2}\gamma_{ijk}x^iy^jz^k$。作为关于$x$的多项式方程，它们可被表示为$f(x)=a_2x^2+a_1x+a_0=0, g(x)=b_2x^2+b_1x+b_0=0$和$h(x)=c_2x^2+c_1x+c_0=0$，其中

$$a_0=\sum_{0\leqslant j+k\leqslant 2}\alpha_{0jk}y^jz^k$$

$$a_1=\sum_{0\leqslant j+k\leqslant 1}\alpha_{1jk}y^jz^k$$

$$a_2=\alpha_{200}$$

$$b_0=\sum_{0\leqslant j+k\leqslant 2}\beta_{0jk}y^jz^k$$

$$b_1=\sum_{0\leqslant j+k\leqslant 1}\beta_{1jk}y^jz^k$$

$$b_2=\beta_{200}$$

$$c_0=\sum_{0\leqslant j+k\leqslant 2}\gamma_{0jk}y^jz^k$$

$$c_1=\sum_{0\leqslant j+k\leqslant 1}\gamma_{1jk}y^jz^k$$

$$c_2=\gamma_{200}$$

$f=0$和$g=0$存在x解的条件为

$$0=(a_2b_1-a_1b_2)(a_1b_0-a_0b_1)-(a_2b_0-a_0b_2)^2=\sum_{0\leqslant i+j\leqslant 4}d_{ij}y^iz^k=:D(y,z)$$

$f=0$和$h=0$存在x解的条件为

$$0=(a_2c_1-a_1c_2)(a_1c_0-a_0c_1)-(a_2c_0-a_0c_2)^2=\sum_{0\leqslant i+j\leqslant 4}e_{ij}y^iz^k=:E(y,z)$$

两个多项式$D(y,z)$和$E(y,z)$都是四次方的。方程$D(y,z)=0$和$E(y,z)=0$可以写成关于y的多项式方程的形式，$d(y)=\sum_{i=0}^{4}d_iy^i$ and $e(y)=\sum_{i=0}^{4}e_iy^i$，其中的系数都是关于$z$的次数为 $\text{degree}(d_i(z))=4-i$ 和$\text{degree}(e_i(z))=4-i$ 的多项式。通过消去y，可以得到一个关于z的多项式。通过贝祖矩阵的行列式来计算d和e，这个 4×4 矩阵$\mathbf{M}=[m_{ij}]$的元素为

$$m_{ij}=\sum_{k=\max(4-j,4-i)}^{\min(4,7-i-j)}w_{k,7-i-j-k}$$

其中 $0\leqslant i\leqslant 3$，$0\leqslant j\leqslant 3$，以及$w_{i,j}=d_ie_j-d_je_i$，其中 $0\leqslant i\leqslant 4$ 和 $0\leqslant j\leqslant 4$。其扩展形式为

$$\mathbf{M}=\begin{bmatrix} w_{4,3} & w_{4,2} & w_{4,1} & w_{4,0} \\ w_{4,2} & w_{3,2}+w_{4,1} & w_{3,1}+w_{4,0} & w_{3,0} \\ w_{4,1} & w_{3,1}+w_{4,0} & w_{2,1}+w_{3,0} & w_{2,0} \\ w_{4,0} & w_{3,0} & w_{2,0} & w_{1,0} \end{bmatrix}$$

$w_{i,j}$的次数为$8-i-j$。贝祖行列式det(M(z))是一个关于z的16次多项式。对于det(M(z))=0的每一个解$\bar{z}$，我们都需要求得对应的$\bar{x}$和$\bar{y}$。

使用贝祖方法隐藏了中间多项式，在前几节的情形中，我们利用这些中间多项式来计算其他的变量。我们将它们明确地找出来。可对$d(y)=d_0+d_1y+d_2y^2+d_3y^3+d_4y^4$和$e(y)=e_0+e_1y+e_2y^2+e_3y^3+e_4y^4$直接应用消元过程。定义$f(y)=e_4d(y)-d_4e(y)=f_0+f_1y+f_2y^2+f_3y^3$。系数为$f_i=e_4d_i-e_id_4$对所有的$i$成立。定义$g(y)=f_3d(y)-d_4yf(y)=g_0+g_1y+g_2y^2+g_3y^3$。系数为$g_0=f_3d_0, g_1=f_3d_1-f_0d_4, g_2=f_3d_2-f_1d_4$和$g_3=f_3d_3-f_2d_4$。现在$f(y)$和$g(y)$都是三次多项式。重复这一过程。定义$h(y)=g_3f(y)-f_3g(y)=h_0+h_1y+h_2y^2$，其中$h_i=g_3f_i-f_3g_i$对所有的$i$成立。定义$m(y)=h_2f(y)-f_3yh(y)=m_0+m_1y+m_2y^2$，其中$m_0=h_2f_0, m_1=h_2f_1-h_0f_3$和$m_2=h_2f_2-h_1f_3$。现在$h(y)$和$m(y)$都是二次多项式。正如我们在前面所看见的，如果多项式有一个公解，它必定是$\bar{y}=(h_2m_0-h_0m_2)/(h_1m_2-h_2m_1)$。由于$d_i$和$e_i$的系数取决于$\bar{z}$，值$h_i$和$m_i$取决于$\bar{z}$。所以，给定$\bar{z}$的一个值，我们可计算一个对应的$\bar{y}$值。$F(x,\bar{y},\bar{z})=a_2x^2+a_1x+a_0=0$和$G(x,\bar{y},\bar{z})=b_2x^2+b_1x+b_0=0$是两个关于$x$的二次方程，可用来计算与特定数对$(\bar{y},\bar{z})$相关的$\bar{x}$。因此，一个公解为$\bar{x}=(a_2b_0-a_0b_2)/(a_1b_2-a_2b_1)$。

【实例】设$F(x,y,z)=(x-1)^2+y^2+z^2-4$（球），$G(x,y,z)=x^2+4y^2-4z$（抛物体）和$H(x,y,z)=x^2+4(y-1)^2+z^2-4$（椭球体）。系数取决于$z$的多项式$d(y)$是$D(y,z)$的一种表示，它的系数为$d_0=-9+40z-10z^2-8z^3-z^4, d_1=0, d_2=-34+24z+6z^2, d_3=0$和$d_4=-9$。系数取决于$z$的多项式$e(y)$是$E(y,z)$的一种表示，它的系数为$(\bar{y},\bar{z})$和$e_4=-9$。矩阵M的$w$项为

$$w_{1,0}=720-3200z+800z^2+640z^3+80z^4$$

$$w_{2,0}=-576+3704z-898z^2-880z^3-122z^4$$

$$w_{3,0}=432-1920z+480z^2+384z^3+48z^4$$

$$w_{4,0}=360z-54z^2-72z^3-9z^4$$

$$w_{2,1}=-2720+1920z+480z^2$$

$$w_{3,1}=0$$

$$w_{3,2}=1632-1152z-288z^2$$

$$w_{4,1}=-720$$

$$w_{4,2}=576+216z+54z^2$$

$$w_{4,3}=-432$$

M的行列式是16次方的，其系数μ_i（$0\leqslant i\leqslant 16$）由下式给出

$$\begin{array}{rlrl} \mu_0 = & 801868087296 & \mu_9 = & 2899639296 \\ \mu_1 = & -4288520650752 & \mu_{10} = & -105691392 \\ \mu_2 = & 4852953907200 & \mu_{11} = & -211071744 \end{array}$$

$$
\begin{array}{llll}
\mu_3 = & -779593973760 & \mu_{12} = & 4082400 \\
\mu_4 = & -1115385790464 & \mu_{13} = & 13856832 \\
\mu_5 = & 307850969088 & \mu_{14} = & 2624400 \\
\mu_6 = & 109063397376 & \mu_{15} = & 209952 \\
\mu_7 = & -34894540800 & \mu_{16} = & 6561 \\
\mu_8 = & -3305131776 & &
\end{array}
$$

$\det(\mathbf{M})(z)=0$的实数根为0.258 255，1.462 02，1.601 99和1.632 58。可以注意到该多项式的系数都较大。为了数值的稳定性，对于每一个z，都应该使用高斯消元法来评估行列式的值，而不应该先去计算μ_i。

三次多项式$f(y)$具有取决于z的系数：

$$f_0=-360z+54z^2+72z^3+9z^4$$

$$f_1=720$$

$$f_2=-576-216z-54z^2$$

$$f_3=432$$

三次多项式$g(y)$具有取决于z的系数：

$$g_0=-3888+17280z-4320z^2-3456z^3+432z^4$$

$$g_1=-3240z+486z^2+648z^3+81z^4$$

$$g_2=-8208+10368z+2592z^2$$

$$g_3=-5184-1944z-486z^2$$

二次多项式$h(y)$具有取决于z的系数：

$$h_0=1679616-5598720z+2286144z^2+1189728z^3-26244z^4-52488z^5-4374z^6$$

$$h_1=-3732480-559872z^2-279936z^3-34992z^4$$

$$h_2=6531840-2239488z-139968z^2+209952z^3+26244z^4$$

三次多项式$m(y)$具有取决于z的系数：

$$m_0=-2351462400z+1158935040z^2+399748608z^3-185597568z^4$$
$$-28343520z^5+15274008z^6+3779136z^7+236196z^8$$

$$m_1=3977330688+806215680z-1088391168z^2-362797056z^3+30233088z^4$$
$$+22674816z^5+1889568z^6$$

$$m_2=-2149908480-120932352z+453496320z^2-37791360z^4-17006112z^5$$
$$-1417176z^6$$

$\bar{z}$值为{0.258 225，1.462 02，1.601 99，1.632 58}。对应的$\bar{y}$值可通过$\bar{y}=(h_2(\bar{z})m_0(\bar{z})-h_0(\bar{z})m_2(\bar{z}))/(h_1(\bar{z})m_2(\bar{z})-h_2(\bar{z})m_1(\bar{z}))$来计算。这些值的有序组为{0.137 429，0.336 959，

1.189 63, 1.149 45}。对应的$\bar{x}$值可通过$\bar{x} = (a_2(\bar{y},\bar{z})b_0(\bar{y},\bar{z}) - a_0(\bar{y},\bar{z})b_2(\bar{y},\bar{z}))/(a_1(\bar{y},\bar{z})b_2(\bar{y},\bar{z}) - a_2(\bar{y},\bar{z})b_1(\bar{y},\bar{z}))$来计算。这些值的有序组为{-0.978 54，2.322 48，0.864 348，1.11596}。这说明，只有$(\bar{x},\bar{y},\bar{z})$的三元组（-0.978 54，0.137 429，0.258 225）和（1.115 96，1.149 45，1.532 58）是真正的交点。其他的两个是额外解（对应的H值分别为 7.187 18 和-0.542 698），这是由于消元过程增大了多项式的次数，因而可能引入与相交无关的根。■

A.3 矩阵分解

我们提供了一些在许多的计算机图形应用中常用的矩阵因式分解方法。Golub 和 Van Load（1993b）是一本很好的涉及矩阵运算的数值方法参考书。Horn 和 Johnson（1985）是一本很好的关于矩阵分析的参考书。

A.3.1 欧拉角度因式分解法

沿坐标轴的旋转是很容易定义和实现的。沿x轴旋转角度为θ的旋转为

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

其中$\theta > 0$表示在平面$x = 0$上的逆时针旋转。假设观察者位于平面上$x > 0$的一侧，并面向原点。

沿y轴旋转角度为θ的旋转为

$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

其中$\theta > 0$表示在平面$y = 0$上的逆时针旋转。假设观察者位于平面上$y > 0$的一侧，并面向原点。

沿z轴旋转角度为θ的旋转为

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

其中$\theta > 0$表示在平面$z = 0$上的逆时针旋转。假设观察者位于平面上$z > 0$的一侧，并面向原点。

沿任意经过原点且单位长度方向为$\hat{u} = (u_x, u_y, u_z)$的轴旋转角度为$\theta$的旋转可表示为

$$\mathbf{R}_{\hat{u}}(\theta) = \mathbf{I} + (\sin\theta)\mathbf{S} + (1 - \cos\theta)\mathbf{S}^2$$

其中I是单位矩阵

$$\mathbf{S} = \begin{bmatrix} 0 & -u_z & u_y \\ u_z & 0 & -u_x \\ -u_y & u_x & 0 \end{bmatrix}$$

并且$\theta>0$表示在平面$\hat{u}\cdot(x,y,z)=0$上的逆时针旋转。假设观察者位于平面上$\hat{u}$所指向的一侧，并面向原点。

1. 旋转矩阵的因式分解

一种常见的方法是将旋转矩阵因式分解为沿坐标轴的旋转的乘积。这种因式分解的形式取决于应用的需要和指定的次序。例如，你可能想要将一个旋转因式分解为$\mathbf{R}=\mathbf{R}_x(\theta_x)\mathbf{R}_y(\theta_y)\mathbf{R}_z(\theta_z)$，其中$\theta_x,\theta_y$和$\theta_z$是一些角度。它的次序是$xyz$。5 种其他的可能次序是$xzy$, yxz, yzx, zxy 和zyx。你可能会想到xyx之类的因式分解，但我们在此不讨论这类形式。下面的讨论使用表示法$c_a=\cos(\theta_a)$ 和$s_a=\sin(\theta_a)$，其中$a=x,y,z$。

2. 形式为$\mathbf{R}_x\mathbf{R}_y\mathbf{R}_z$的因子

设$\mathbf{R}=[r_{ij}]$，其中 $0\leqslant i\leqslant 2$ 且 $0\leqslant j\leqslant 2$，用$\mathbf{R}_x(\theta_x)\mathbf{R}_y(\theta_y)\mathbf{R}_z(\theta_z)$与其相乘，得到如下的方程

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix}=\begin{bmatrix} c_yc_z & -c_ys_z & s_y \\ c_zs_xs_y+c_xs_z & c_xc_z-s_xs_ys_z & -c_ys_x \\ -c_xc_zs_y+s_xs_z & c_zs_x+c_xs_ys_z & c_xc_y \end{bmatrix}$$

从中可得$s_y=r_{02}$，因此有$\theta_y=\sin^{-1}(r_{02})$。如果$\theta_y\in(-\pi/2,\pi/2)$，则$c_y\neq 0$，$c_y(s_x,c_x)=(-r_{12},r_{22})$，此时有$\theta_x=\operatorname{atan2}(-r_{12},r_{22})$。类似地，$c_y(s_z,c_z)=(-r_{01},r_{00})$，此时有$\theta_z=\operatorname{atan2}(-r_{01},r_{00})$。

如果$\theta_y=\pi/2$，则有$s_y=1$和$c_y=0$。此时有

$$\begin{bmatrix} r_{10} & r_{11} \\ r_{20} & r_{21} \end{bmatrix}=\begin{bmatrix} c_zs_x+c_xs_z & c_xc_z-s_xs_z \\ -c_xc_z+s_xs_z & c_zs_x+c_xs_z \end{bmatrix}=\begin{bmatrix} \sin(\theta_z+\theta_x) & \cos(\theta_z+\theta_x) \\ -\cos(\theta_z+\theta_x) & \sin(\theta_z+\theta_x) \end{bmatrix}$$

因此，$\theta_z+\theta_x=\operatorname{atan2}(r_{10},r_{11})$。存在一个自由因子，所以因式分解不是惟一的。一种选择是$\theta_z=0$和$\theta_x=\operatorname{atan2}(r_{10},r_{11})$。

如果$\theta_y=-\pi/2$，则$s_y=-1$和$c_y=0$。此时有

$$\begin{bmatrix} r_{10} & r_{11} \\ r_{20} & r_{21} \end{bmatrix}=\begin{bmatrix} -c_zs_x+c_xs_z & c_xc_z+s_xs_z \\ c_xc_z+s_xs_z & c_zs_x-c_xs_z \end{bmatrix}=\begin{bmatrix} \sin(\theta_z-\theta_x) & \cos(\theta_z-\theta_x) \\ \cos(\theta_z-\theta_x) & -\sin(\theta_z-\theta_x) \end{bmatrix}$$

因此$\theta_z-\theta_x=\operatorname{atan2}(r_{10},r_{11})$。存在一个自由因子，所以因式分解不是惟一的。一种选择是$\theta_z=0$和$\theta_x=-\operatorname{atan2}(r_{10},r_{11})$。

这种因式分解的伪码为

```
thetaY = asin(r02);
if (thetaY < PI/2) {
    if (thetaY > -PI / 2) {
        thetaX = atan2(-r12, r22);
        thetaZ = atan2(-r01, r00);
    } else {
        // not a unique solution (thetaX - thetaZ constant)
        thetaX = -atan2(r10, r11);
        thetaZ = 0;
    }
} else {
```

```
        // not a unique solution (thetaX + thetaZ constant)
        thetaX = atan2(r10, r11);
        thetaZ = 0;
    }
```

3. 形式为$\mathbf{R}_x\mathbf{R}_z\mathbf{R}_y$的因子

设$\mathbf{R}=[r_{ij}]$，其中$0\leqslant i\leqslant 2$且$0\leqslant j\leqslant 2$，用$\mathbf{R}_x(\theta_x)\mathbf{R}_z(\theta_z)\mathbf{R}_y(\theta_y)$与其相乘，得到如下的方程

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} c_yc_z & -s_z & c_zs_y \\ s_xs_y+c_xc_ys_z & c_xc_z & -c_ys_x+c_xs_ys_z \\ -c_xs_y+c_ys_xs_z & c_zs_x & c_xc_y+s_xs_ys_z \end{bmatrix}$$

方法分析与xyz的情形类似，其伪码为

```
thetaZ = asin(-r01);
if (thetaZ < PI / 2) {
    if (thetaZ > -PI / 2) {
        thetaX = atan2(r21, r11);
        thetaY = atan2(r02, r00);
    } else {
        // not a unique solution (thetaX + thetaY constant)
        thetaX = atan2(-r20, r22);
        thetaY = 0;
    }
} else {
    // not a unique solution (thetaX - thetaY constant)
    thetaX = atan2(r20, r22);
    thetaY = 0;
}
```

4. 形式为$\mathbf{R}_y\mathbf{R}_x\mathbf{R}_z$的因子

设$\mathbf{R}=[r_{ij}]$，其中$0\leqslant i\leqslant 2$且$0\leqslant j\leqslant 2$，用$\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)\mathbf{R}_z(\theta_z)$与其相乘，得到如下的方程

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} c_yc_z+s_xs_ys_z & c_zs_xs_y-c_ys_z & c_xs_y \\ c_xs_z & c_xc_z & -s_x \\ -c_zs_y+c_ys_xs_z & c_yc_zs_x+s_ys_z & c_xc_y \end{bmatrix}$$

方法分析与xyz的情形类似，其伪码为

```
thetaX = asin(-r12);
if (thetaX < PI / 2) {
    if (thetaX > -PI / 2) {
        thetaY = atan2(r02, r22);
        thetaZ = atan2(r10, r11);
    } else {
        // not a unique solution (thetaY + thetaZ constant)
        thetaY = atan2(-r01, r00);
        thetaZ = 0;
    }
} else {
    // not a unique solution (thetaY - thetaZ constant)
    thetaY = atan2(r01, r00);
    thetaZ = 0;
}
```

5. 形式为$R_yR_zR_x$的因子

设$\mathbf{R}=[r_{ij}]$，其中$0\leqslant i\leqslant 2$且$0\leqslant j\leqslant 2$，用$\mathbf{R}_y(\theta_y)\mathbf{R}_z(\theta_z)\mathbf{R}_x(\theta_x)$与其相乘，得到如下的方程

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} c_yc_z & s_xs_y - c_xc_ys_z & c_xs_y + c_ys_xs_z \\ s_z & c_xc_z & -c_zs_x \\ -c_zs_y & c_ys_x + c_xs_ys_z & c_xc_y - s_xs_ys_z \end{bmatrix}$$

方法分析与xyz的情形类似，其伪码为

```
thetaZ = asin(r10);
if (thetaZ < PI / 2) {
    if (thetaZ > -PI / 2) {
        thetaY = atan2(-r20, r00);
        thetaX = atan2(-r12, r11);
    } else {
        // not a unique solution (thetaX - thetaY constant)
        thetaY = -atan2(r21, r22);
        thetaX = 0;
    }
} else {
    // not a unique solution (thetaX + thetaY constant)
    thetaY = atan2(r21, r22);
    thetaX = 0;
}
```

6. 形式为$R_zR_xR_y$的因子

设$\mathbf{R}=[r_{ij}]$，其中$0\leqslant i\leqslant 2$且$0\leqslant j\leqslant 2$，用$\mathbf{R}_z(\theta_z)\mathbf{R}_x(\theta_x)\mathbf{R}_y(\theta_y)$与其相乘，得到如下的方程

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} c_yc_z - s_xs_ys_z & -c_xs_z & c_zs_y + c_ys_xs_z \\ c_zs_xs_y + c_ys_z & c_xc_z & -c_yc_zs_x + s_ys_z \\ -c_xs_y & s_x & c_xc_y \end{bmatrix}$$

方法分析与xyz的情形类似，其伪码为

```
thetaX = asin(r21);
if (thetaX < PI / 2) {
    if (thetaX > -PI / 2) {
        thetaZ = atan2(-r01, r11);
        thetaY = atan2(-r20, r22);
    } else {
        // not a unique solution (thetaY - thetaZ constant)
        thetaZ = -atan2(r02, r00);
        thetaY = 0;
    }
} else {
    // not a unique solution (thetaY + thetaZ constant)
    thetaZ = atan2(r02, r00);
    thetaY = 0;
}
```

7. 形式为$\mathbf{R}_z\mathbf{R}_y\mathbf{R}_x$的因子

设$\mathbf{R}=[r_{ij}]$，其中$0\leqslant i\leqslant 2$且$0\leqslant j\leqslant 2$，用$\mathbf{R}_z(\theta_z)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)$与其相乘，得到如下的方程

$$\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix} = \begin{bmatrix} c_yc_z & c_zs_xs_y - c_xs_z & c_xc_zs_y + s_xs_z \\ c_ys_z & c_xc_z + s_xs_ys_z & -c_zs_x + c_xs_ys_z \\ -s_y & c_ys_x & c_xc_y \end{bmatrix}$$

方法分析与xyz的情形类似，其伪码为

```
thetaY = asin(-r20);
if (thetaY < PI / 2) {
    if (thetaY > -PI / 2) {
        thetaZ = atan2(r10, r00);
        thetaX = atan2(r21, r22);
    } else {
        // not a unique solution (thetaX + thetaZ constant)
        thetaZ = atan2(-r01, -r02);
        thetaX = 0;
    }
} else {
    // not a unique solution (thetaX - thetaZ constant)
    thetaZ = -atan2(r01, r02);
    thetaX = 0;
}
```

A.3.2 QR分解法

给定一个$n\times n$的可逆矩阵$\mathbf{A}$，我们希望将其分解为$\mathbf{A}=\mathbf{QR}$，其中$\mathbf{Q}$是一个正交矩阵，$\mathbf{R}$是上三角矩阵。这种因式分解其实就是对$\mathbf{A}$的各列应用格拉姆－施密特正交化算法。设这些列表示为$\vec{a}_i$，其中$1\leqslant i\leqslant n$。由于假设$\mathbf{A}$是可逆的，因此这些列是线性无关的。格拉姆－施密特算法从该向量集构建出一个正交集$\hat{q}_i$，$1\leqslant i\leqslant n$。即每一个向量都是单位向量，并且向量都是互相垂直的。

第一步很简单，只需规整化$\hat{q}_1$，即

$$\hat{q}_1=\frac{\vec{a}_1}{\|\vec{a}_1\|}$$

我们可以将$\vec{a}_2$投影到$\hat{q}_1$的正交补空间上，并表示为$\vec{a}_2=c_1\hat{q}_1+\vec{p}_2$，其中$\hat{q}_1\cdot\vec{p}_2=0$。将该方程与$\hat{q}_1$进行点积，得到$c_1=\hat{q}_1\cdot\vec{a}_2$。进行改写，可得$\vec{p}_2=\vec{a}_2-(\hat{q}_1\cdot\vec{a}_2)\hat{q}_1$。通过构建，向量$\hat{q}_1$和$\vec{p}_2$是互相垂直的，但是$\vec{p}_2$并不需要是单位长度的。因此，定义$\hat{q}_2$为规整化的$\vec{p}_2$：

$$\hat{q}_2=\frac{\vec{a}_2-(\hat{q}_1\cdot\vec{a}_2)\hat{q}_1}{\|\vec{a}_2-(\hat{q}_1\cdot\vec{a}_2)\hat{q}_1\|}$$

对$\vec{a}_3$进行相似的构建。我们可将$\vec{a}_3$投影到$\hat{q}_1$和$\hat{q}_2$生成的子空间的正交补空间上，并表示为$\vec{a}_3=c_1\hat{q}_1+c_2\hat{q}_2+\vec{p}_3$，其中$\hat{q}_1\cdot\vec{p}_3=0$且$\hat{q}_2\cdot\vec{p}_3=0$。将该方程与$\hat{q}_i$进行点积，得到$c_i=\hat{q}_i\,\vec{a}_3$。进行改写，可得$\vec{p}_3=\vec{a}_3-(\hat{q}_1\cdot\vec{a}_3)\hat{q}_1-(\hat{q}_2\cdot\vec{a}_3)\hat{q}_2$。该正交集中的下一个向量是规整化的$\vec{p}_3$：

$$\hat{q}_3=\frac{\vec{a}_3-(\hat{q}_1\cdot\vec{a}_3)\hat{q}_1-(\hat{q}_2\cdot\vec{a}_3)\hat{q}_2}{\|\vec{a}_3-(\hat{q}_1\cdot\vec{a}_3)\hat{q}_1-(\hat{q}_2\cdot\vec{a}_3)\hat{q}_2\|}$$

一般地，对于$i\geqslant 2$，

$$\hat{q}_i=\frac{\vec{a}_i-\sum_{j=1}^{i-1}(\hat{q}_j\cdot\vec{a}_i)\hat{q}_j}{\|\vec{a}_i-\sum_{j=1}^{i-1}(\hat{q}_j\cdot\vec{a}_i)\hat{q}_j\|}$$

设Q为该$n\times n$矩阵，它的各列为向量$\hat{q}_i$。因式分解中的上三角矩阵为

$$\mathbf{R}=\begin{bmatrix}\hat{q}_1\cdot\vec{a}_1 & \hat{q}_1\cdot\vec{a}_2 & \cdots & \hat{q}_1\cdot\vec{a}_n\\ 0 & \hat{q}_2\cdot\vec{a}_2 & \cdots & \hat{q}_2\cdot\vec{a}_n\\ \vdots & \vdots & & \vdots\\ 0 & 0 & \cdots & \hat{q}_n\cdot\vec{a}_n\end{bmatrix}$$

A.3.3 特征值分解法

给定一个$n\times n$的矩阵A，其特征系统（eigensystem）的形式为$\mathrm{A}\vec{x}=\lambda\vec{x}$或者$(\mathrm{A}-\lambda\mathrm{I})\vec{x}=\vec{0}$。要求存在解$\vec{x}\neq\vec{0}$。为了满足这样的条件，矩阵$\mathrm{A}-\lambda\mathrm{I}$必须是不可逆的。即满足$\det(\mathrm{A}-\lambda\mathrm{I})=0$，这是一个关于λ的$n$次多项式，叫做A的特征多项式（characteristic polynomial）。对每一个根λ，计算矩阵$\mathrm{A}-\lambda\mathrm{I}$，求解系统$(\mathrm{A}-\lambda\mathrm{I})\vec{x}=\vec{0}$的非零解。标准的线性代数课本用符号来表示进行这种操作的大量例子，但是大多数的实际应用需要进行这种处理的健壮的数值方法。特别地，如果$n\geqslant 5$，不存在求解多项式的根的近似公式，因此必须应用数值方法。Press 等（1998）是关于求解特征系统的一本很好的参考书。

大多数要求特征系统的图形应用都具有对称矩阵。数值方法非常适合于这类应用，因为一个完整的特征向量的基底总是存在。标准的方式是应用正交变换，这类正交变换称为豪斯霍尔德变换（Householder transformation），来将A简化为三对角矩阵。递归应用这种QR方法，将三对角矩阵简化为对角矩阵。Press 等（1998）建议使用一种具有隐含移位的QL算法，以使得算法尽可能地健壮。对于$n=3$，通过简单地计算$\det(\mathrm{A}-\lambda\mathrm{I})=0$的根就能求解问题。可以避免我们多次提到过的数值问题，因为最终的结果是一些数据的视觉展示，数值误差在其中显得并不如在高精度的应用中重要。

求解一个$n\times n$的对称矩阵A的特征系统隐含了对A进行因式分解。设从λ_1至λ_m为A的相异的特征值，λ_i的特征空间的维数为$d_i\geqslant 1$，并且$\sum_{i=1}^{m}d_i=n$。即我们可以选择向量$\hat{q}_i$，其中$1\leqslant i\leqslant n$的一个正交集，也就是每一个正交空间的正交集的并集。由于这些向量本身就是特征向量，$\mathrm{A}\hat{q}_i=\lambda_{i'}\hat{q}_i$，其中$\lambda_{i'}$是对应于$\hat{q}_i$所在的特征空间的特征值。定义对角矩阵$\Lambda=\mathrm{Diag}\{\lambda_{1'},\ldots,\lambda_{n'}\}$，定义Q为列为$\hat{q}_i$的矩阵。特征方程可合并写成$\mathrm{AQ}=\mathrm{Q}\Lambda$或者$\mathrm{A}=\mathrm{Q}\Lambda\mathrm{Q}^{\mathrm{T}}$。最后一个方程叫做A的特征分解（eigendecomposition）。

A.3.4 极分解法

假设一个对象通过一定次序的齐次矩阵运算进行了平移、旋转和非均衡缩放。总的变换就是各个变换的乘积。初学者经常问的一个问题是，如何从总的变换中抽取出其中的平移、旋转和非均衡缩放变换。这个问题提得不合逻辑，但是产生这个问题的原因如下。假

设齐次变换如下式中的中括号内的形式

$$H_i = \left[\begin{array}{c|c} \mathrm{R}_i\mathrm{S}_i & \vec{t}_i \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right]$$

其中$\vec{t}_i$是一个3×1平移向量；$\vec{0}^{\mathrm{T}}$是一个1×3零向量；S_i是齐次缩放矩阵，一个3×3对角矩阵，其对角项都为正；R_i是一个3×3旋转矩阵。假设对一个对象进行了n个这样的变换，最后的变换为$H = H_n H_{n-1} \cdots H_2 H_1$。问题的核心是对下式进行因式分解

$$H = \left[\begin{array}{c|c} \mathbf{RS} & \vec{t} \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right]$$

显然，平移向量是很容易被选出的。然而，缩放和旋转比较棘手。仅仅考虑两个这种齐次变换的乘积

$$\left[\begin{array}{c|c} \mathbf{RS} & \vec{t} \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right] = \left[\begin{array}{c|c} \mathrm{R}_2\mathrm{S}_2 & \vec{t}_2 \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} \mathrm{R}_1\mathrm{S}_1 & \vec{t}_1 \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right] = \left[\begin{array}{c|c} \mathrm{R}_2\mathrm{S}_2\mathrm{R}_1\mathrm{S}_1 & \mathrm{R}_2\mathrm{S}_2\vec{t}_1 + \vec{t}_2 \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right]$$

总的变换是$\vec{t} = \mathrm{R}_2\mathrm{S}_2\vec{t}_1 + \vec{t}_2$。总的缩放－旋转分量是$\mathrm{R}_2\mathrm{S}_2\mathrm{R}_1\mathrm{S}_1$。在$\mathrm{S}_1$和$\mathrm{S}_2$都表示均衡缩放的情形中，即$\mathrm{S}_i = \sigma_i \mathrm{I}$，其中$i=1$，2，$\mathbf{I}$是单位矩阵，我们可以惟一地确定$\mathbf{R}$和$\mathbf{S}$如下

$$\mathrm{R}_2\mathrm{S}_2\mathrm{R}_1\mathrm{S}_1 = \mathrm{R}_2\sigma_2\mathrm{I}\mathrm{R}_1\sigma_1\mathrm{I} = \mathrm{R}_2\sigma_2\mathrm{R}_1\sigma_1 = \mathrm{R}_2\mathrm{R}_1(\sigma_2\sigma_1)$$

σ_2和R_1可进行交换只是矩阵的数量乘法的一个性质。最后的选择是$\mathbf{S} = \sigma_2\sigma_1\mathbf{I}$，另一个均衡缩放，以及$\mathbf{R} = \mathrm{R}_2\mathrm{R}_1$，这是一个旋转，因为它是两个旋转的乘积。在两个齐次项的特殊情形中，如果S_2是均衡缩放矩阵，那么$\mathrm{S} = \mathrm{S}_2\sigma_1$且$\mathrm{R} = \mathrm{R}_2\mathrm{R}_1$。但是，当$\mathrm{S}_2$是非均衡的时，问题就变得复杂了。一般地，如果$\mathbf{D}_1$是一个对角矩阵，它的对角项中至少有两个不相同，并且如果$\mathbf{R}_1$是一个旋转矩阵，那么不可能找到一个对角矩阵$\mathbf{D}_2$，使其满足$\mathbf{R}_1\mathbf{D}_1 = \mathbf{D}_2\mathbf{R}_2$，其中$\mathbf{R}_2$是一个旋转矩阵。一般地，如果图形引擎的变换系统允许非均衡缩放，就会使问题变得复杂化。

直观的问题是，非均衡缩放是沿着特定的轴进行的，因此，这些轴的方向是非常重要的因素。考虑在二维空间中圆$x^2 + y^2 = 1$的旋转和非均衡旋转变换。图 A.1 示意了这一问题。注意观察，如果我们运行沿着不同的坐标轴集进行缩放，在该例中，轴为$(1,1)/\sqrt{2}$和$(-1,1)/\sqrt{2}$，我们可以迫使在下面的圆最终伸展为上面的椭圆。沿着$(1,1)/\sqrt{2}$方向的缩放比例必须为2。

这样就得到了被称为极分解的方法（polar decomposition）。通过将指定系统旋转为标准坐标系统，可以获得在指定坐标系统中的非均衡缩放，在标准系统进行缩放，再旋转回到原系统中。如果$\mathbf{R}$表示从指定坐标系统到原系统的旋转，并且$\mathbf{D}$是在标准坐标系统中的非均衡缩放对角矩阵，那么在指定坐标系统中的缩放为$\mathbf{S} = \mathbf{R}^{\mathrm{T}}\mathbf{D}\mathbf{R}$。这只是用数学公式来陈述我们刚才用语言所做的说明。矩阵$\mathbf{S}$要求是对称的。

给定一个矩阵$\mathbf{A}$，其极分解为$\mathbf{A} = \mathbf{QS}$，其中$\mathbf{Q}$是一个正交矩阵，而$\mathbf{S}$是一个对称矩阵。在前面提到的应用中，我们感兴趣的矩阵是$\mathbf{A} = \mathrm{R}_2\mathrm{S}_2\mathrm{R}_1\mathrm{S}_1$。一般不可能进行因式分解$\mathbf{A} = \mathbf{RS}$，其中$\mathbf{R}$是一个旋转，而$\mathbf{S}$是一个对角矩阵。极分解总是可能的。对称矩阵$\mathbf{S}$表示在某些坐标系统中的缩放。如果坐标系统是一个标准系统（方向为$(1,0,0)$, $(0,1,0)$, $(0,0,1)$），那么$\mathbf{S}$是

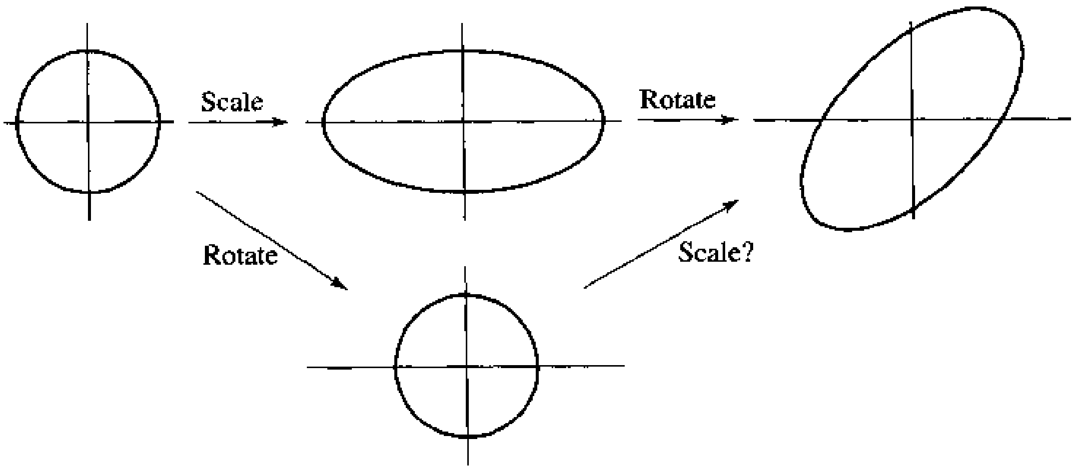


图 A.1　顶部的次序显示首先进行一个不均匀比例缩放$(x, y) \to (2x, y)$，然后逆时针旋转$\pi/4$弧度。底部的次序显示一次任意角度旋转（圆在旋转时没有任何变化），但清楚地表明没有顶部次序中的使圆变成椭圆的沿坐标轴的不均衡比例缩放

对角的。注意观察$A^TA = S^TQ^TQS = S^TS = S^2$，其中第二个等式成立是因为Q是正交的，而第三个等式成立是因为S是对称的。矩阵A^TA是正半无穷的，因为$\vec{x}^TB\vec{x} = \vec{x}^TA^TA\vec{x} = \|A\vec{x}\|^2 \geq 0$。因此$S^2 = A^TA$必须有一个正半无穷的平方根（Horn 和 Johnson 1985）。通过特征分解$S^2 = R^TDR$（其中R是正交的，D是具有非负对角项的对角矩阵），可以得到该平方根。一个平方根为$S = R^TD^{1/2}R$，其中$D^{1/2}$为对角矩阵，其对角项为D的对角项的平方根。如果S是可逆的（缩放都是正的），通过$Q = AS^{-1}$可以获得D。如果S是不可逆的，那么A也是不可逆的。这类矩阵的极分解可以通过奇异值分解来实现，我们将在下一节讨论这类问题。在典型的图形应用中，A实际上是可逆的。

上一段中说明的构建S的方法当然是进行分解的一种有效方法。如果使用了一种标准的数值特征系统求解方法，则这种构建方法是递归的。另一种也是递归的方法，首先构建Q，由 Ken Shoemake 在 Heckbert（1994）中提出。一旦已知Q，$S = Q^TA$。递归起始点是$Q_0 = A$。下一次递归由$Q_{i+1} = (Q_i + Q_i^{-T})/2$产生。当连续递归之间的变换足够小时，递归结束。

A.3.5　奇异值分解法

特征分解自然地将对称矩阵A因式分解为$A = R^TDR$，其中R是正交的，而D是对角的。对于非对称矩阵，并不是总能进行这类因式分解的。可能进行的因式分解称为奇异值分解（singular value decomposition）。任何矩阵A都能因式分解为$A = LSR^T$，其中L和R都是正交矩阵，而S是具有非负对角项的对角矩阵（证明参见 Horn 和 Johnson 1985）。S的对角项是AA^T的特征值，L的列是AA^T的特征向量，而R的列是A^TA的特征向量，L和R的列相对于特征值的排列次序于是一致。结果是，由A^TA和AA^T的特征分解可以产生奇异值分解。然而，实际上还有进行分解的更高效的数值方法，例如，Golub 和 Van Load（1993）中提出的方法。

A.4 三维旋转表示法

本节讨论三种不同的表示三维旋转的方法，即矩阵方法、轴一角方法和四元数方法，以及如何进行不同方法之间的相互转换。

A.4.1 矩阵表示法

一个二维旋转是如下形式的变换

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \end{bmatrix}$$

其中 θ 是旋转角。一个三维旋转是在经过原点的特定平面内的二维旋转。这样旋转可用 3×3 旋转矩阵 $\mathbf{R} = [\hat{r}_0\, \hat{r}_1\, \hat{r}_2]$ 来表示，它的列 $\hat{r}_0, \hat{r}_1$ 和 $\hat{r}_2$ 构成一个右手正交集。因此，$\|\hat{r}_0\| = \|\hat{r}_1\| = \|\hat{r}_2\| = 1$，$\hat{r}_0 \cdot \hat{r}_1 = \hat{r}_0 \cdot \hat{r}_2 = \hat{r}_1 \cdot \hat{r}_2 = 0$ 和 $\hat{r}_0 \cdot \hat{r}_1 \times \hat{r}_2 = 1$。该矩阵的列对应于标准基底向量 $(1, 0, 0), (0, 1, 0)$ 和 $(0, 0, 1)$ 的最终旋转值，次序也相同。给定一个 3×1 向量 $\vec{x} = [x_j]$ 和一个 3×3 旋转矩阵 $\mathbf{R} = [r_{ij}]$，旋转向量为

$$\mathbf{R}\vec{x} = \left[\sum_{j=0}^{2} r_{ij} x_j \right] \tag{A.3}$$

A.4.2 轴一角表示法

如果旋转的平面具有单位长度法线 $\hat{w}$，那么旋转的轴一角表示就是数对 $\langle \hat{w}, \theta \rangle$。旋转的方向选择为从 $\hat{w}$ 所指向的一面向下看时，旋转是沿着原点逆时针方向旋转 $\theta > 0$。这与二维空间旋转的惯例相同。

1. 轴一角到矩阵的转换

如果 $\hat{u}$, $\hat{v}$ 和 $\hat{w}$ 构成一个右手正交系，那么任意点都可表示为 $\vec{x} = u_0\hat{u} + v_0\hat{v} + w_0\hat{w}$。参见 9.2.2 节关于与平面相关的坐标系的内容。$\vec{x}$ 关于轴 $\hat{w}$ 且旋转角为 θ 的旋转满足 $\mathbf{R}\vec{x} = u_1\hat{u} + v_1\hat{v} + w_1\hat{w}$。在几何意义上很清楚，$w_1 = w_0 = \hat{w} \cdot \vec{x}$。另外两个分量的变换就像对它们施行了一次二维旋转，因此 $u_1 = \cos(\theta)u_0 - \sin(\theta)v_0$ 和 $v_1 = \sin(\theta)u_0 + \cos(\theta)v_0$。利用正交系的右手法则，可以很容易证明

$$\hat{w} \times \vec{x} = u_0\hat{w} \times \hat{u} + v_0\hat{w} \times \hat{v} + w_0\hat{w} \times \hat{w} = -v_0\hat{u} + u_0\hat{v}$$

和

$$\hat{w} \times (\hat{w} \times \vec{x}) = -v_0\hat{w} \times \hat{u} + u_0\hat{w} \times \hat{v} = -u_0\hat{u} - v_0\hat{w}$$

按显示的形式合并它们，并利用 u_0, v_0, u_1, 和 v_1 之间的关系，可得

$$\begin{aligned}(\sin\theta)\hat{w} \times \vec{x} + (1 - \cos\theta)\hat{w} \times (\hat{w} \times \vec{x}) = {} & (-v_0 \sin\theta - u_0(1 - \cos\theta))\hat{u} \\ & + (u_0 \sin\theta - v_0(1 - \cos\theta))\hat{v}\end{aligned}$$

$$= (u_1 - u_0)\hat{u} + (v_1 - v_0)\hat{v}$$

$$= R\vec{x} - \vec{x}$$

因此，$\vec{x}$ 关于给定轴 $\hat{w}$ 旋转角为 θ 的旋转为

$$R\vec{x} = \vec{x} + (\sin\theta)\hat{w} \times \vec{x} + (1-\cos\theta)\hat{w} \times (\hat{w} \times \vec{x}) \tag{A.4}$$

通过定义下式，上式也可写成矩阵形式，其中 $\hat{w} = (a, b, c)$,

$$S = \begin{bmatrix} 0 & -c & b \\ c & 0 & -a \\ -b & a & 0 \end{bmatrix}$$

其中

$$R = I + (\sin\theta)S + (1-\cos\theta)S^2$$

于是有

$$R\vec{x} = \vec{x} + (\sin\theta)S\vec{x} + (1-\cos\theta)S^2\vec{x}$$

2. 矩阵到轴—角的转换

相反的问题是，从旋转矩阵出发，抽取旋转角度和单位长度的轴。有多种解，因为当 $\hat{w}$ 是有效的轴时，$-\hat{w}$ 也是有效的轴，并且当 θ 是有效的角时，$\theta + 2\pi k$ 也是有效的角。首先，将一个矩阵的迹（trace）定义为其所有对角项之和。用代数方法可以证明 $\cos\theta = (\text{Trace}(R) - 1)/2$，其中

$$\theta = \cos^{-1}((\text{Trace}(R) - 1)/2) \in [0, \pi] \tag{A.5}$$

还很容易证明

$$R - R^{T} = (2\sin\theta)S \tag{A.6}$$

其中S是一个斜对称矩阵。下面的构建方法假设 $\theta = 0, \theta \in (0, \pi)$ 和 $\theta = \pi$。

如果 $\theta = 0$，那么任何单位长度方向都可作为有效的轴，因为没有进行旋转。如果 $\theta \in (0, \pi)$，那么可从方程（A.6）直接抽取轴，$\vec{d} = (r_{21} - r_{12}, r_{02} - r_{20}, r_{10} - r_{01})$ 和 $\hat{w} = \vec{d}/\|\vec{d}\|$。如果 $\theta = \pi$，由于 $R - R^{T} = 0$，因此方程（A.6）对于构建轴并没有帮助。在这种情形中，注意

$$R = I + 2S^2 = \begin{bmatrix} 1 - 2(w_1^2 + w_2^2) & 2w_0w_1 & 2w_0w_2 \\ 2w_0w_1 & 1 - 2(w_0^2 + w_2^2) & 2w_1w_2 \\ 2w_0w_2 & 2w_1w_2 & 1 - 2(w_0^2 + w_1^2) \end{bmatrix}$$

其中 $\hat{w} = (w_0, w_1, w_2)$。其基本思想是从旋转矩阵的对角项中抽取轴的最大分量。如果 r_{00} 是最大值，那么 w_0 必须是在数量上的最大分量。计算 $4w_0^2 = r_{00} - r_{11} - r_{22} + 1$，并选取 $w_0 = \sqrt{r_{00} - r_{11} - r_{22} + 1}/2$。其结果是，$w_1 = r_{01}/(2w_0)$ 和 $w_2 = r_{02}/(2w_0)$。如果 r_{11} 是最大值，那么计算 $4w_1^2 = r_{11} - r_{00} - r_{22} + 1$ 并选取 $w_1 = \sqrt{r_{11} - r_{00} - r_{22} + 1}/2$。其结果是，$w_0 = r_{01}/(2w_1)$ 和 $r_{12}/(2w_1)$。最后，如果 r_{22} 是最大值，那么计算 $4w_2^2 = r_{22} - r_{00} - r_{11} + 1$ 并选取 $w_2 = \sqrt{r_{22} - r_{00} - r_{11} + 1}/2$。其结果是，$w_0 = r_{02}/(2w_2)$ 和 $w_1 = r_{12}/(2w_2)$。

A.4.3 四元数表示法

第三种表示法涉及单位四元数（unit quaternion）。这里仅提供这种表示法的简介。关于旋转和四元数之间关系的详细内容可从 Shoemake（1987）和 Eberly（2000）中找到。单位四元数可表示为$q = w + xi + yj + zk$，其中w, x, y 和z是实数，而四元组(w, x, y, z)是单位长度。单位四元数的集合是$\mathbb{R}^4$上的单位超球面。i，j 和 k 之间的乘积定义为$i^2 = j^2 = k^2 = -1, ij = -ji = k, jk = -kj = i$，以及$ki = -ik = j$。注意，这些乘积不具有交换性。两个单位四元数的乘积$q_n = w_n + x_n i + y_n j + z_n k$（$n$=0，1）用对和的分配积来定义，注意操作数的次序是很重要的：

$$\begin{aligned} q_0 q_1 = &(w_0 w_1 - x_0 x_1 - y_0 y_1 - z_0 z_1) \\ &+ (w_0 x_1 + x_0 w_1 + y_0 z_1 - z_0 y_1) i \\ &+ (w_0 y_1 - x_0 z_1 + y_0 w_1 + z_0 x_1) j \\ &+ (w_0 z_1 + x_0 y_1 - y_0 x_1 + z_0 w_1) k \end{aligned}$$

q的共轭定义为

$$q^* = w - xi - yj - zk$$

注意，$qq^* = q^*q = 1$，其中右边的 1 是四元数的w项，其x，y和z项都为零。

1．轴－角到四元数的转换

如果$\hat{a} = (x_0, y_0, z_0)$是旋转的单位长度轴，而且$\theta$是旋转角，则表示该旋转的四元数$q = w + xi + yj + zk$满足$w = \cos(\theta/2)$，$x = x_0 \sin(\theta/2), y = y_0 \sin(\theta/2)$和$z = z_0 \sin(\theta/2)$。

如果一个向量$\vec{v} = (v_0, v_1, v_2)$表示为四元数$v = v_0 i + v_1 j + v_2 k$，并且如果$q$表示一个旋转，那么旋转向量$\vec{u}$可以表示为四元数$u = u_0 i + u_1 j + u_2 k$，其中

$$u = qvq^* \tag{A.7}$$

可以证明，u的w项实际上为 0。

2．四元数到轴－角的转换

设$q = w + xi + yj + zk$为一个单位四元数。如果$\|w\| = 1$，那么其旋转角为$\theta = 0$，并且任意单位长度的方向向量都可作为其旋转轴，因为没有进行旋转。

如果$\|w\| < 1$，可得其旋转角为$\theta = 2\cos^{-1}(w)$，并且其旋转轴可由下式算出：$\hat{u} = (x, y, z)/\sqrt{1 - w^2}$。

3．四元数到矩阵的转换

利用恒等式$2\sin^2(\theta/2) = 1 - \cos(\theta)$和$\sin(\theta) = 2\sin(\theta/2)\cos(\theta/2)$可以很容易地证明$2wx = (\sin\theta)w_0$, $2wy = (\sin\theta)w_1$, $2wz = (\sin\theta)w_2$, $2x^2 = (1 - \cos\theta)w_0^2$, $2xy = (1 - \cos\theta)w_0 w_1$, $2xz = (1 - \cos\theta)w_0 w_2$, $2y^2 = (1 - \cos\theta)w_1^2$，$2yz = (1 - \cos\theta)w_1 w_2$和$2z^2 = (1 - \cos\theta)w_2^2$。这些等式的右项都是表达式$\mathrm{R} = \mathrm{I} + (\sin\theta)\mathrm{S} + (1 - \cos\theta)\mathrm{S}^2$中的项。替换它们，可得

$$R = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2wz & 2xz-2wy \\ 2xy-2wz & 1-2x^2-2z^2 & 2yz-2wx \\ 2xz+2wy & 2yz-2wx & 1-2x^2-2y^2 \end{bmatrix} \quad (A.8)$$

4. 矩阵到四元数的转换

我们在前面提到过$\cos\theta = (\text{Trace}(R)-1)/2$。利用恒等式$2\cos^2(\theta/2) = 1+\cos\theta$，可得$w^2 = \cos^2(\theta/2) = (\text{Trace}(R)+1)/4$或者$|w| = \sqrt{\text{Trace}(R)+1}/2$。如果 $\text{Trace}(R) > 0$，那么$|w| > 1/2$，因此，不失一般性，选取w为正的平方根，$w = \sqrt{\text{Trace}(R)+1}/2$。恒等式$R - R^T = (2\sin\theta)S$可产生$(r_{21}-r_{12}, r_{02}-r_{20}, r_{10}-r_{01}) = 2\sin\theta(w_0, w_1, w_2)$。最后，前面导出的恒等式为$2xw = w_0\sin\theta$, $2yw = w_1\sin\theta$，以及$2zw = w_2\sin\theta$。将它们组合起来可得$x = (r_{21}-r_{12})/(4w)$, $y = (r_{20}-r_{02})/(4w)$和$z = (r_{10}-r_{01})/(4w)$。

如果$\text{Trace}(R) \leqslant 0$，那么$|w| \leqslant 1/2$。其要点是首先从方程（A.8）中的旋转R的对角项中抽取x、y或z中最大的，如果r_{00}是最大的对角项，那么在数量上x要大于y或z。通过代数运算可以证明，$4x^2 = r_{00}-r_{11}-r_{22}+1$，从中可以选取$x = \sqrt{r_{00}-r_{11}-r_{22}+1}/2$。其结果是，$w = (r_{12}-r_{21})/(4x)$, $y = (r_{01}+r_{10})/(4x)$和$z = (r_{02}+r_{20})/(4x)$。如果r_{11}是最大的对角项，那么计算$4y^2 = r_{11}-r_{00}-r_{22}+1$并选取$y = \sqrt{r_{11}-r_{00}-r_{22}+1}/2$。其结果是，$w = (r_{20}-r_{02})/(4y)$, $x = (r_{01}+r_{10})/(4y)$和$z = (r_{12}+r_{21})/(4y)$。最后，如果r_{22}是最大的对角项，那么计算$4z^2 = r_{22}-r_{00}-r_{11}+1$，并选取$z = \sqrt{r_{22}-r_{00}-r_{11}+1}/2$。其结果是，$w = (r_{01}-r_{10})/(4z)$, $x = (r_{02}+r_{20})/(4z)$和$y = (r_{12}+r_{21})/(4z)$。

A.4.4 性能问题

经常被提及的一个问题是，“使用旋转的最优表示法是什么？”与大多数计算机科学的话题一样，这个问题并没有答案，只有折中的考虑。在下面的讨论中，旋转矩阵为**R**，四元数为q，而轴－角对为$(\hat{a}, \theta)$。不同的高级运算与包括乘法(M)，加法或减法(A)，除法(D)以及费时的数学库函数评测(F)等的一些低级运算进行对比。在实际的实现中，比较(C)也应该计算在内，因为它们可能甚至比乘法和（或）加法更费时。我们提供了一个总结表，使你能快速地比较它们的性能。

1. 内存的使用

一个旋转矩阵要求 9 个浮点数，一个四元数要求 4 个浮点数，一个轴－角数对要求 4 个浮点数，因此，显然旋转矩阵需要更多的内存。当要求变换时，仅仅存储轴－角公式中的角显然是没有用的，因为还需要知道$\sin\theta$和$1-\cos\theta$的值。计算三角函数是非常费时的。最好是预先将这两个值计算出来并保存起来，因此，实际上轴－角数对将要求 6 个浮点数。因此，四元数是使用内存最少的方法。表 A.1 是内存使用情况的总结。轴－角方法包含 3 个记录轴的浮点数，1 个记录角θ的浮点数，以及 2 个记录$\sin\theta$和$1-\cos\theta$的浮点数。如果不计算余弦三角函数的值，那么任何要求这类函数值的运算都是非常费时的。

表 A.1 使用内存的比较

表示法	浮点数数目	说　明
旋转矩阵	9	
轴—角数对	4	不预先计算 $\sin\theta$ 和 $1-\cos\theta$
轴—角数对	6	预先计算 $\sin\theta$ 和 $1-\cos\theta$
四元数	4	

2. 转换时间

经常使用旋转的应用总要从一种旋转表示法转换为另一种旋转表示法，因此，评测转换所需的时间是非常有用的。涉及的实体包括一个旋转矩阵R，一个轴—角数对$(\hat{a},\theta)$和一个四元数q。假设旋转角的范围为$(0,\pi)$。

（1）轴－角到矩阵的转换

计算$\sigma=\sin(\theta)$和$\gamma=\cos(\theta)$需要进行两次函数调用。计算项$1-\gamma$需要一次加法。从$\hat{a}$中获得的斜对称矩阵S不要求任何运算。矩阵S^2要求 6 次不同的乘法和 3 次加法，没有计算符号的变化。计算项$(1-\gamma)S^2$要求 6 次不同的乘法。计算项σS要求 3 次不同的乘法。最后，组合$R=I+\sigma S+(1-\gamma)S^2$使用了 9 次加法。总的花费是$13A+15M+2F$。

（2）矩阵到轴－角的转换

抽取$\theta=\cos^{-1}((\mathrm{Trace}(R)-1)/2)$需要 3 次加法，1 次乘法和 1 次函数调用。向量$\vec{d}=(r_{21}-r_{12},r_{02}-r_{20},r_{10}-r_{01})$要求 3 次加法。规整化向量$\hat{a}=\vec{d}/|\vec{d}|$要求 6 次乘法，2 次加法，1 次除法和 1 次函数调用。总的花费是$8A+7M+1D+2F$。

（3）轴－角到四元数的转换

抽取$\theta=2\cos^{-1}(w)$需要 1 次函数调用和 1 次乘法。构建$\hat{a}=(x,y,z)/\sqrt{1-w^2}$需要 4 次乘法，1 次除法，1 次加法和 1 次函数调用。总的花费是$1A+5M+1D+2F$。

（4）四元数到矩阵的转换

计算$\theta/2$需要一次乘法。计算$\sigma=\sin(\theta/2)$和$w=\cos(\theta/2)$需要 2 次函数调用。计算乘积$(x,y,z)=\sigma\hat{a}$要求 3 次乘法。总的花费是 $4M+2F$。

（5）四元数到矩阵的转换

这种转换要求 12 次乘法。计算项$t_x=2x$，$t_y=2y$和$t_z=2z$。基于它们计算下列的项：$t_{wx}=wt_x$，$t_{wy}=wt_y$，$t_{wz}=wt_z$，$t_{xx}=t_xx$，$t_{xy}=xt_y$，$t_{xz}=xt_z$，$t_{yy}=t_yy$，$t_{yz}=yt_z$和$t_{zz}=t_zz$。计算旋转矩阵的项需要 12 次加法：$r_{00}=1-t_{yy}-t_{zz}$，$r_{01}=t_{xy}-t_{wz}$，$r_{02}=t_{xz}+t_{wy}$，$r_{10}=t_{xy}+t_{wz}$，$r_{11}=1-t_{xx}-t_{zz}$，$r_{12}=t_{yz}-t_{wx}$，$r_{20}=t_{xz}-t_{wy}$，$r_{21}=t_{yz}+t_{wx}$，以及$r_{22}=1-t_{xx}-t_{yy}$。总的花费是$12A+12M$。

（6）矩阵到四元数的转换

这种转换取决于R的迹的符号。计算迹$\tau=\mathrm{Trace}(R)$需要 2 次加法。假设$\tau>0$（该比较的时间花费为 $1C$）。计算$w=\sqrt{\tau+1}/2$要求 1 次加法，1 次乘法和 1 次函数调用。计算表达式$\lambda=1/(4w)$要求 1 次乘法和 1 次除法。计算项$x=\lambda(r_{21}-r_{12})$，$y=\lambda(r_{02}-r_{20})$和$z=\lambda(r_{10}-r_{01})$需要 3 次加法和 3 次乘法。总的花费是$6A+5M+1D+1F+1C$。

如果$\tau \leqslant 0$，那么必须找到旋转矩阵的对角项中的最大值。这要求两次比较，其调用花费是 $2C$。为了讨论起来方便，假设r_{00}是最大值。计算$x=\sqrt{r_{00}-r_{11}-r_{22}+1}/2$要求 3 次加法，1 次乘法和 1 次函数调用。计算表达式$\lambda=1/(4x)$要求 1 次乘法和 1 次除法。计算项$w=\lambda(r_{21}-r_{12}), y=\lambda(r_{10}+r_{01})$和$z=\lambda(r_{20}+r_{02})$要求 3 次加法和 3 次乘法。总的花费是 $6A+5M+1D+1F+3C$。

表 A.2 总结了不同的旋转表示法之间转换所需的时间花费。

表 A.2　不同旋转表示法之间的转换运算结果的比较

转　换	A	M	D	F	C
轴一角到矩阵	13	15		2	
矩阵到轴一角	8	7	1	2	
轴一角到四元数	1	5	1	2	
四元数到轴一角		4		2	
四元数到矩阵	12	12			
矩阵到四元数（$\tau>0$）	6	5	1	1	1
矩阵到四元数（$\tau\leqslant 0$）	6	5	1	1	3

3. 变换时间

用旋转矩阵来变换$\vec{v}$就是进行乘积$\vec{u}=\mathrm{R}\vec{v}$，这要求 9 次乘法和 6 次加法，总共需要 15 次运算。

如果$\vec{v}=(v_0,v_1,v_2)$且$v=v_0i+v_1j+v_2k$对应于w分量为零的四元数，那么旋转向量$\vec{u}=(u_0,u_1,u_2)$就是计算$u=u_0i+u_1j+u_2k=qvq^*$。对四元数乘法直接应用一般公式，乘积$p=qv$要求 16 次乘法和 12 次加法。乘积pq^*也要求相同数量的运算。总的运算次数是 56。然而，由于v没有w项，p只要求 12 次乘法和 8 次加法，其中一项理论上为零，因此不需要计算它。我们也知道u没有w项，因此乘积pq^*只要求 12 次乘法和 9 次加法。利用这些优化措施，总的运算次数是 41。注意，从四元数q到旋转矩阵R要求 12 次乘法和 12 次加法。用R来变换$\vec{v}$要进行 15 次运算。因此，转换到旋转的处理过程和相乘使用了 39 次运算，比计算qvq^*少 2 次。当变换大量的顶点集时，实现四元数库并且仅使用四元数的纯化主义者将会悲哀地浪费大量的时间周期。

利用轴一角数对来变换$\vec{v}$的公式如下

$$\mathrm{R}\vec{v}=\vec{v}+(\sin\theta)\hat{a}\times\vec{v}+(1-\cos\theta)\hat{a}\times(\hat{a}\times\vec{v})$$

正如我们在前面所提及的，除了存储轴和角的值之外，还应该预先计算出$\sin\theta$和$1-\cos\theta$，并存储它们的值，总共需要 6 个浮点数。叉积$\hat{a}\times\hat{v}$使用 6 次乘法和 3 次加法。$\hat{a}\times(\hat{a}\times\hat{v})$也一样，假设先计算出括号内的叉积，并存储为临时变量。用一个数量与叉积相乘需要 6 次乘法。将三个向量相加需要 6 次加法。因此，我们需要 18 次乘法和 12 次加法，总共需要 30 次运算。

因此，旋转公式能实现更快的变换。对于一个向量来说，四元数公式实现的变换最慢。但是，应该注意，对于n个向量的一组变换，仅仅需要进行一次从四元数到旋转矩阵的转换，这种转化需要 24 次运算。利用四元数的变换的全部运算次数为 $24+15n$。轴一角公式

要用 30n 次运算，因此，对于两个或更多的向量来说，四元数变换更快一些。表 A.3 总结了变换一个向量的运算次数。表 A.4 总结了变换 n 个向量的变换次数。

表 A.3 变换一个向量的运算结果的比较

表示法	A	M	说明
旋转矩阵	6	9	
轴一角	12	18	
四元数	24	32	使用基础的四元数乘法
四元数	17	24	使用特殊的四元数乘法
四元数	18	21	先转换为矩阵，再相乘

表 A.4 变换换 n 个向量的运算结果的比较

表示法	A	M	说明
旋转矩阵	6n	9n	
轴一角	12n	18n	
四元数	24n	32n	使用基础的四元数乘法
四元数	17n	24n	使用特殊的四元数乘法
四元数	12+6n	12+9n	先转换为矩阵，再相乘

4. 组合

两个旋转矩阵的乘积需要 27 次乘法和 18 次加法，共总需要 18A+27M 次运算。

两个四元数的乘积需要 16 次乘法和 12 次加法，总共需要 12A+16M 次运算，显然优于矩阵乘法。而且，重新规整化一个四元数以校正浮点误差比利用格拉姆一施密特正交化方法重新校正一个旋转矩阵花费的时间更少。

对于要求计算效率的应用来说，组合两个轴一角数对是不可想像的。实现这种组合的一种方法是先转换为矩阵，将矩阵相乘，然后再抽取轴一角数对。从轴一角到矩阵的两次转换的代价是 26A+30M+4F，矩阵乘法的代价是 18A+27M，而从矩阵到轴一角的转换的代价是 8A+7M+1D+2F。总的代价是 52A+64M+1D+6F。

实现两个轴一角数对的组合的另一种方法是转换为四元数，将四元数相乘，再抽取轴一角数对。从轴一角到四元数的两次转换的代价是 2A+10M+2D+4F，四元数乘法的代价是 12A+16M，而从四元数到轴一角的转换的代价是 4M+2F。总的代价是 14A+30M+2D+6F。表 A.5 总结了组合两次旋转的运算次数。

表 A.5 组合运算结果的比较

表示法	A	M	D	F
旋转矩阵	18	27		
四元数	12	16		
轴一角（转换为矩阵）	52	64	1	6
轴一角（转换为四元数）	14	30	2	6

5. 插值

本节将讨论如何对旋转进行插值的问题，我们将分别讨论三种不同的表示法的情形。

（1）四元数插值

四元数非常适合于插值。进行插值的标准方法是球面线性插值（spherical linear interpolation），简称为 slerp。给定两个夹角为锐角 θ 的四元数 p 和 q，slerp 定义为 $s(t;p,q) = p(p^*q)^t$（$t\in[0,1]$）。注意，$s(0;p,q)=p$ 且 $s(1;p,q)=q$。slerp 的一种等价定义更便于计算，即

$$s(t;p,q)=\frac{\sin((1-t)\theta)p+\sin(t\theta)q}{\sin(\theta)}$$

如果将 p 和 q 看成是一个单位圆上的点，上述的公式就是它们之间的最短弧的参数表示。如果一个点根据该参数表示沿着曲线运动，那么它以常速运动。这样，任何位于[0,1]内的 t 的均衡采样都会得到均匀分布于弧上的点。

我们假设仅仅指定了 p，q 和 t。并且，由于 q 和$-q$ 表示相同的旋转，如果需要保证被看成是四元组的 p 和 q 之间的夹角是锐角，就可以用$-q$ 还替换 q。也就是 $p\cdot q\geqslant 0$。作为四元组，p 和 q 都是单位长度的。因此，它们之间的点积为 $p\cdot q=\cos(\theta)$。表 A.6 说明了运算次数。对于显示在左边的包括一个已经计算好的项的各项，仅仅计算了其加法运算，以避免重复计算运算次数。

表 A.6 四元数插值运算结果

项	A	M	D	F
$a_0=p\cdot q$	3	4		
$\theta=\cos^{-1}(a_0)$				1
$1-t$	1			
$(1-t)\theta$		1		
$t\theta$		1		
$\sin(\theta)$				1
$\sin((1-t)\theta)$				1
$\sin(t\theta)$				1
$a_1=1/\sin(\theta)$			1	
$a_2=a_1\sin((1-t)\theta)$		1		
$a_3=a_1\sin(t\theta)$		1		
a_2p+a_3q	4	8		
总计	8	16	1	4

（2）旋转矩阵插值

认为四元数法优于旋转矩阵法的一个借口是（旋转矩阵法）缺少一个可以直接用于旋转矩阵的、有意义的插值公式。然而，可以用一种效果与 slerp 等价的方法来对旋转矩阵进行插值。如果 **P** 和 **Q** 是对应于四元数 p 和 q 的旋转矩阵，则这两个矩阵的 slerp 是

$$S(t;P,Q)=P(P^TQ)^t$$

这是定义四元数的 slerp 的同一个公式。其中的技术问题是，定义R^t对于旋转矩阵 **R** 和实数值 t 的意义。如果旋转具有旋转轴$\hat{a}$和旋转角θ，那么R^t具有相同的旋转轴，但是旋转角为$t\theta$。计算旋转矩阵的 slerp 的过程如下：

① 计算$R=P^TQ$。

② 从 **R** 中抽取旋转轴$\hat{a}$和旋转角θ。

③ 通过转换轴一角数对$\hat{a}$，$t\theta$ 来计算R^t。

④ 计算$S(t;P,Q)=PR^t$。

这种算法要求进行一次轴一角抽取，这涉及一次反三角函数调用和一次平方根计算，一系列的三角计算（计算$t\theta$），以及一次回到旋转矩阵的转换。与仅仅需要三次三角调用的四元数的 slerp 相比，这是相当费时的。因此，四元数插值的效率更高，但是对于希望在应用中避免四元数的纯粹主义者来说，这确实是一种对旋转矩阵进行插值的方法。

表 A.7 显示了运算次数，其格式和规则与显示四元数插值的表 A.6 相同。四元数和旋转矩阵插值都使用 1 次除法和 4 次函数调用。然而，与四元数插值相比，旋转矩阵所需的加法和乘法的次数较多。

表 A.7 旋转矩阵插值运算结果

项	*A*	*M*	*D*	*F*
$R=P^TQ$	18	27		
$a_0=0.5(\text{Trace}(R)-1)$	4	1		
$\theta=\cos^{-1}(a_0)$				1
$\vec{d}=(r_{21}-r_{12},r_{02}-r_{20},r_{10}-r_{01})$	3			
$a_1=1/\lvert\vec{d}\rvert$	2	3	1	1
$\hat{a}=a_1\vec{d}$		3		
$t\theta$		1		
$a_2=\sin(t\theta)$				1
$a_3=1-\cos(t\theta)$	1			1
Matrix S, no cost				
S^2	3	6		
$R^t=I+a_2S+a_3S^2$	9	9		
PR^t	18	27		
总计	58	77	1	4

（3）轴一角插值

不存在明显和自然的方法来对轴一角表示进行与四元数和旋转矩阵类似的插值。惟一的方法是将轴一角表示转换为四元数或者旋转矩阵，进行插值，然后将插值结果变换回轴一角形式。与旋转的组合一样，这是非常费时的方法。

A.5 求根

给定一个连续的函数 $\vec{F}: D \subset \mathbb{R}^n \to \mathbb{R}^n$，我们要做的是寻找一个 $\vec{x}$（或者找到一个点集），使其满足 $\vec{F}(\vec{x}) = 0$。

A.5.1 一维方法

给定一个连续的函数 $f:[a,b] \to \mathbb{R}$，第一个问题是，是否存在一些满足 $f(r)=0$ 的 $r \in [a,b]$。如果 $f(a)f(b) < 0$，那么至少存在一个根。然而，也可能存在多个根。如果计算得到了根 r，就要求进行另外的分析来求取其余的根。例如，如果 f 是一个多项式且 r 是一个根，该函数可进行因式分解为 $f(t) = (t-r)^p g(t)$，其中 $p \geqslant 1$ 且 g 是 $\text{degree}(g) = \text{degree}(f) - p$ 的一个多项式。求根的过程转化为在 $[a, b]$ 上求函数 g 的根。

如果 $f(a)f(b) > 0$，那么不能保证 f 在 $[a, b]$ 内存在根。对于这类问题，可以先确定根的范围。将区间分区成 $t_i = a + i(b-a)/n$，其中 $0 \leqslant i \leqslant n$。如果对于某些 i 有 $f(t_i)f(t_{i+1}) < 0$，那么将该子区间进行对分以定位一个根。对 n 的合理选取与应用程序对其函数 f 的了解有关。

最后，可能需要计算 $f:\mathbb{R} \to \mathbb{R}$ 的根，其中 f 的定义域并不是一个有界区间。可以在 $[-1,1]$ 内寻找 f 的根。位于该区间之外的任何 f 的根 t 都可用 $t=1/r$ 来计算，其中 $r \in [-1, 1]$ 是 $g(r) = f(1/r)$ 的一个根。

1. 对分法

对分法是求解一个连续函数 $f:[a,b] \to \mathbb{R}$ 的一个根的方法，先确定一个包含根的区间，然后不断地对该区间进行对分以逼近根。假定初始时有 $f(a)f(b) < 0$。由于 f 是连续的，所以必定存在一个根 $r \in (a,b)$。区间的中点是 $m = (a+b)/2$。计算该点的函数值 $f(m)$，并将其与端点的函数值进行比较。如果 $f(a)f(m) < 0$，那么一个根位于子区间 (a, m) 内，然后再对该子区间进行对分处理。如果 $f(m)f(b) < 0$，那么一个根位于子区间（m，b）内，然后再对该子区间进行对分处理。如果 $f(m) = 0$，或者在指定的公差范围内为零，那么终止处理。当前子区间的长度也可以作为停止的条件，即如果子区间的长度足够小，则也应该终止算法。如果在 [a，b] 内存在一个根，那么对分法一定能找到它。然而，收敛的速度很慢。

2. 牛顿法

给定一个可微的函数 $f:\mathbb{R} \to \mathbb{R}$，首先估计函数在何处的值为零，即 $(x_0, f(x_0))$。利用函数图形在该点的切线来更新该点的估计位置，（希望）获得更接近的一个点。函数的切线是 $y - f(x_0) = f'(x_0)(x - x_0)$，它与 x 轴相交于 $(0, x_1)$，因此，$-f(x_0) = f'(x_0)(x_1 - x_0)$。假设 $f'(x_0) \neq 0$，求解 x_1，可得

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

迭代的下一点是 $(x_1, f(x_1))$，重复进行该过程直到满足一个终止条件，一般可以得到一

个函数值接近于零的点。与对分法不同，牛顿法并不能保证递归是收敛的，但是如果存在收敛，则收敛的速度较快。成功与否取决于对 x_0的初始估计值。

3. 多项式的根

给定一个次数为 n 的多项式 $f(t)=\sum_{i=0}^{n}a_it^n$，其中 $a_n\neq 0$。可以使用标准的求根方法来求解该多项式的根，但是利用该函数的性质可以得到一种更好的方法。对于 $2\leqslant n\leqslant 4$，存在一个形式接近的方程，可以用它来求解多项式的根。可以直接利用公式来求解，但是可能产生数值问题，特别是当多项式具有多个大于1的根时，问题更严重。例如，一个二次多项式 $f(t)=at^2+b^t+c$ 的根为 $t=(-b\pm\sqrt{b^2-4ac})/(2a)$。如果 $b^2-4ac=0$，则该二次多项式具有一个双根 $t=-b/(2a)$。然而，数值的舍入误差可能导致 $b^2-4ac=-\epsilon<0$，且 ϵ 是一个非常小的数。导致数值问题的另一种情形是当 a 接近于零时。如果出现这种情况，则可以求解 $g(t)=t^2f(1/t)=ct^2+bt+a=0$ 得到 $t=(-b\pm\sqrt{b^2-4ac})/(2c)$。然而，如果 c 也接近于零，则问题依然存在。对于三次和四次多项式也存在类似的问题。

有一种基于递归的方法可以用一种方法来使每一个区间刚好包含一个根。对于每一个这样的区间，可以利用对分法来求得区间内的根。也可以使用一种综合了对分法步骤和牛顿法步骤的混合方法。只有当牛顿法步骤产生的递归位于当前区间范围之外时才使用对分法步骤。这种方法的好处是利用牛顿法的快速收敛来求根，但是如果牛顿法不收敛，则使用对分法来为牛顿递归产生一个更好的初始估值。

（1）导数次序的边界根

求解划界问题的一种简单方法是将 $\mathbb{R}$ 分解为一些区间，使多项式 $f(t)$ 在每一个区间内都是单调的。如果可以确定该多项式的导数在何处为零，该集合就提供了相应的分解。如果 d_i 和 d_{i+1} 是满足 $f'(d_i)=f'(d_{i+1})=0$ 的连续值，那么必定有在 (d_i,d_{i+1}) 上 $f'(t)>0$ 或者 (d_i,d_{i+1}) 上 $f'(t)<0$。在任何一种情形中，f 在该区间内都最多只可能有一个根。条件 $f(d_i)f(d_{i+1})<0$，$f(d_i)=0$ 或者 $f(d_{i+1})=0$ 用于确定这个根的存在性。

求解 $f'(t)=0$ 所需的技术与求解 $f(t)=0$ 的技术相同。它们之间的差别是 $\text{degree}(f')=\text{degree}(f)-1$。递归实现可以确保该问题的求解，其基本情形是永远不为零或者恒等于零的常数多项式。

如果 $f'(t)\neq 0$（$t\in(-\infty,d_0)$），那么 f 可能在半无穷区间 $(-\infty,d_0]$ 内存在一个根。不能利用对分法来确定这个根，因为该区间是无界的。然而，可以确定一个包含该多项式的根的最大有界区间。这种确定方法建立在谱半径（spectral radius）或者矩阵的范数（norm of a matrix）的概念之上（参见 Horn 和 Johnson 1985）。给定一个方阵 $\mathbf{A}$，其谱半径记为 $\rho(\mathbf{A})$，就是矩阵 $\mathbf{A}$ 的特征值的绝对值中的最大值。矩阵 $\mathbf{A}$ 的范数，记为 $\|\mathbf{A}\|$，就是必须满足如下5个条件的数量值函数：当且仅当 $\mathbf{A}=0$ 时 $\|\mathbf{A}\|\geqslant 0$，$\|\mathbf{A}\|=0$，对于任何数量 c 都有 $\|cA\|=|c|\|\mathbf{A}\|$，$\|\mathbf{A}+\mathbf{B}\|\leqslant\|\mathbf{A}\|+\|\mathbf{B}\|$，以及 $\|\mathbf{AB}\|\leqslant\|\mathbf{A}\|\|\mathbf{B}\|$。谱半径和任意矩阵范数之间的关系为 $\rho(\mathbf{A})\leqslant\|\mathbf{A}\|$。给定 $f(t)=\sum_{i=0}^{n}a_it^i$，其中 $a_n=1$，则伴随矩阵为

$$\mathbf{A}=\begin{bmatrix}-a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0\\ 1 & 0 & \cdots & 0 & 0\\ 0 & 1 & \cdots & 0 & 0\\ \vdots & \vdots & \ddots & \vdots & \vdots\\ 0 & 0 & \cdots & 1 & 0\end{bmatrix}$$

其特征多项式为 $f(t)=\det(\mathrm{A}-t\mathrm{I})$，因此 f 的根为A的特征值。谱范数因而提供了根的范围。由于有许多的矩阵范数可供选择，因此存在许多种可能的范围。其中的一种范围是柯西范围：

$$|t|\leqslant\max\{|a_0|,1+|a_1|,\ldots,1+|a_{n-1}|\}=1+\max\{|a_0|,\ldots,|a_{n-1}|\}$$

可以得到的另一种范围是卡麦克尔和梅森范围：

$$|t|\leqslant\sqrt{1+\sum_{i=0}^{n-1}|a_i|^2}$$

如果 $a_0\neq 0$，那么 $f(0)\neq 0$，因此 f 的根的范围不包括零。通过使用 $g(t)=[t^n f(1/t)]/a_0$ 可以构建较小的范围。$g(t)$ 的根为 $f(t)$ 的根的倒数。对 $g(t)$ 应用柯西范围，然后取倒数，可得

$$|t|\geqslant\frac{|a_0|}{1+\max\{1,|a_1|,\ldots,|a_{n-1}|\}}$$

卡麦克尔和梅森范围为

$$|t|\geqslant\frac{|a_0|}{\sqrt{1+\sum_{i=0}^{n-1}|a_i|^2}}$$

将这些范围用于递归调用，以确定 $f(t)$ 在何处是单调的。该多项式可被因式分解为 $f(t)=t^p g(t)$，其中 $p\geqslant 0$，并且 g 是满足 $g(0)\neq 0$ 的多项式。如果 $p=0$，那么 $f=g$，并且在 $0<a\leqslant|t|\leqslant b$ 内处理 f，其中 a 和 b 是从前面提到的不等式中计算得到的范围。如果 $p>0$，那么在利用同一不等式求得的范围所得的区间内处理 g。

（2）斯图姆次序的边界根

考虑定义在区间 $[a,b]$ 内的多项式 $f(t)$。f 的一个斯图姆次序就是一个多项式 $f_i(t)$，（$0\leqslant i\leqslant m$）的集合，它们满足 $\mathrm{degree}(f_{i+1})>\mathrm{degree}(f_i)$，并且 f 在 $[a,b]$ 内的根是 $N=s(a)-s(b)$，其中 $s(a)$ 是 $f_0(a),\ldots,f_m(a)$ 的符号改变的次数，而 $s(b)$ 是 $f_1(b),\ldots,f_m(b)$ 的符号改变的次数。f 在 $\mathbb{R}$ 上的实数根的总数为 $s(-\infty)-s(\infty)$。$m=\mathrm{degree}(f)$ 并不总是成立。

著名的斯图姆次序是 $f_0(t)=f(t), f_1(t)=f'(t)$，以及 $f_i(t)=-\mathrm{Remainder}(f_{i-2}/f_{i-1})$（$i\geqslant 2$）。用这种方法产生多项式，直到剩余的项为常数。摘自D. G. Hook和P. R. McAree（1990）中的一个范例是 $f(t)=t^3+3t^2-1$。其斯图姆次序为 $f_0(t)=t^3+3t^2-1, f_1(t)=3t^2+6t, f_2(t)=2t+1$ 和 $f_3=9/4$。表 A.8 列出了不同的 t 值的斯图姆多项式的符号。设 $N(a,b)$ 表示在区间 (a,b) 内的实数根的个数，该表显示了 $N(-\infty,-3)=0$，$N(-3,-2)=1$，$N(-2,-1)=0$，$N(-1,0)=1$，$N(0,1)=1$，以及 $N(1,\infty)=0$。并且，负实数根的个数为 $N(-\infty,0)=2$，正实数根的个数为 $N(0,\infty)=1$，总的实数根个数为 $N(-\infty,\infty)=3$。

下面的例子说明了斯图姆次序中的多项式的数量并不需要为 $\mathrm{degree}(f)+1$。函数 $f(t)=(t-1)^3$ 有一个斯图姆次序 $f_0(t)=(t-1)^3, f_1(t)=3(t-1)^2$，以及 $f_2(t)\equiv 0$，这是因为 f_1 刚好可被 f_0 整除。表 A.9 列出了 f 在不同的 t 值上的符号变化。实数根的总数为 $N(-\infty,\infty)=1$。

表 A.8 斯图姆多项式 t^3+3t^2-1 在 t 取不同的值时的符号

t	$f_0(t)$ 的符号	$f_1(t)$ 的符号	$f_2(t)$ 的符号	$f_3(t)$ 的符号	符号变化
$-\infty$	−	+	−	+	3
−3	−	+	−	+	3
−2	+	0	−	+	2
−1	+	−	−	+	2
0	−	0	+	+	1
+1	+	+	+	+	0
$+\infty$	+	+	+	+	0

表 A.9 斯图姆多项式 $(t-1)^3$ 在 t 取不同的值时的符号

t	$f_0(t)$ 的符号	$f_1(t)$ 的符号	$f_2(t)$ 的符号	符号变化
$-\infty$	−	+	0	1
0	−	+	0	1
$+\infty$	+	+	0	0

A.5.2 多维方法

本节讨论对分法和牛顿法在多维上的扩展。

1. 对分法

一维的对分法可以扩展到多维。设 $(f,g):[a,b]\times[c,d]\to\mathbb{R}^2$。现在的问题是寻找一个满足 $(f(x,y),g(x,y))=(0,0)$ 的点 $(x,y)\in[a,b]\times[c,d]$。可以利用 $[a,b]\times[c,d]$ 的四叉树分解来搜索根。从初始矩形开始搜索，在其四个顶点上计算 f 和 g。

- 如果 f 或者 g 之一在四个顶点上具有相同的符号，则算法停止处理该区域。
- 如果 f 和 g 都在四个顶点上改变符号，则在区域的中心计算它们。如果在中心处的值足够接近于零，则该点作为一个根返回，并且停止在该区域上的搜索。
- 如果在中心处的值不足够接近于零，则利用原来的四个顶点、四条边的中点和中心将该区域分解为四个子区域。将算法递归应用于这四个子区域。

可能出现下面的情形：由于 f 或 g 在四个顶点上都具有相同的符号而不对一个区域进行进一步的处理，但是该区域实际上却包含一个根。这与一维的情形类似——初始的矩阵需要分解，以确定根位于哪一个子矩形范围内。可将对分法应用于每一个至少包含一个根的子矩形。

对于三维情形，用相似的方法来应用三叉树分解。对于 n 维的情形，则使用 2^n 叉树分解。

2. 牛顿法

给定一个可微的函数 $F:\mathbb{R}^n\to\mathbb{R}^n$，可以利用一维牛顿法的扩展来求解方程 $F(\vec{x})=\vec{0}$。直接概括这种方法的迭代方式是选取一个初始估值 $(\vec{x}_0, F(\vec{x}_0))$，并用下式来产生下一次迭代：

$$\vec{x}_1 = \vec{x}_0 - (\mathrm{D}F(\vec{x}_0))^{-1}F(\vec{x}_0)$$

数量$\mathrm{D}F(\vec{x})$为F的偏微分矩阵，称为雅克比矩阵，并包含项$\partial F_i/\partial x_j$，其中$F_i$是$F$的第$i$个分量，而$x_j$是$\vec{x}$的第$j$个分量。每一次迭代都要求一次矩阵反转。但是这种明显的扩展经常并非最好用的方法。有一些这种方法的变体在实用性方面更好用，其中一些使用了所谓的“分裂法”（splitting method），这种方法能避免反转矩阵，并且经常具有更好的收敛性。

A.5.3 二次方程的稳定解

二次方程的一般形式为

$$ax^2 + bx + c = 0$$

并且其解为

$$\lambda = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

然而，在实现计算机程序时，简单地用直观的方法来求解该方程可能导致数值问题。如果我们对标准混解进行变换，就能看出其中的问题（Casselman 2001）：

$$\{x_0, x_1\} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$$= \left(\frac{b}{2a}\right)\left(-1 \pm \sqrt{1 - \frac{4ac}{b^2}}\right)$$

如果数量$\frac{4ac}{b^2}$非常小，那么其中的一个解将涉及两个几乎相等的整数的减法，这可能导致较大的舍入误差。

可是，如果我们改写二次方程，并因式分解出a：

$$ax^2 + bx + c = a(x^2 + \frac{b}{a}x + \frac{c}{a})$$

$$= a(x - x_0)(x - x_1)$$

那么，我们就可看出两个根的乘积为$\frac{c}{a}$。通过做如下次序的运算就能避免上述的问题：

$$A \leftarrow \frac{b}{2a}$$

$$B \leftarrow \frac{4ac}{b^2}$$

$$\leftarrow \frac{c}{aA^2}$$

$$C \leftarrow -1 - \sqrt{1 - B}$$

$$x_0 \leftarrow AC$$

$$x_1 \leftarrow \frac{AB}{C}$$

A.6 最小值

关于最小值的基本问题就是寻找函数 $f: D \subset \mathbb{R}^n \to \mathbb{R}$ 的全局最小值的问题。求函数的最小值时，要求该函数至少是连续的，并且假设 D 为一个紧凑集。如果该函数是连续可微的，那么这样的事实有助于确定最小值出现的位置，但是也存在一些不需要通过求导就能计算最小值的方法。

A.6.1 一维方法

考虑 $f: [t_{\min}, t_{\max}] \to \mathbb{R}$。如果 f 是可微的，那么全局最小值必定出现于一个满足 $f' = 0$ 的点或者出现于区间的一个端点。这种标准方法就是应用于计算一个点与一条线段之间的距离的方法（参见 6.3.1 节关于求解距离的方法的讨论）。该距离平方函数是一个二次函数并且定义在一个紧凑区间上。该函数的最小值出现在该区间的一个内点（最接近点是线段的一个内点）或者出现在一个端点。求解问题 $f'(t) = 0$ 可能使问题复杂化。这是一个 A.5 节所描述的求根问题。

Brent 法

连续而不一定可微的函数必定在一个紧凑集上取得一个最小值。在计算最小值时，不需要求导或者当函数可微时不需要确定何处导数为零是非常理想的。这样的一种方法叫做 Brent 法，它以迭代方式来应用反抛物线插值。

其基本思想是用三个值 $(t_0, f(t_0)), (t_m, f(t_m))$ 和 $(t_1, f(t_1))$（$t_{\min} \leqslant t_0 < t_m < t_1 \leqslant t_{\max}$），其中 $f(t_m) < f(t_0)$ 且 $f(t_m) < f(t_1)$，来界定最小值。这意味着函数在 $t \in [t_0, t_m]$ 上的一些值必须递减，并且在 $t \in [t_m, t_1]$ 上的一些值必须递增。这样就保证该函数在 $[t_0, t_1]$ 内的某处具有局部最小值。Brent 法试图缩小最小值的范围，与对分法缩小函数的根的范围很相似（参见 A.5 节）。

下面的方法是 Press 等（1988）中描述的 Brent 法的一种变体。三个界定点位于一条抛物线 $p(t)$ 上。该抛物线的顶点一定位于 (t_0, t_1) 内。设 $f_0 = f(t_0)$, $f_m = f(t_m)$ 和 $f_1 = f(t_1)$。该抛物线的顶点出现在 $t_v \in (t_0, t_1)$，并且可被表示为

$$t_v = t_m - \frac{1}{2}\frac{(t_1 - t_0)^2(f_0 - f_m) - (t_0 - t_m)^2(f_1 - f_m)}{(t_1 - t_m)(f_0 - f_m) - (t_0 - t_m)(f_1 - f_m)}$$

在该处计算函数值，$f_v = f(t_v)$。如果 $t_v < t_m$，那么新的界定点为 $(t_0, f_0), (t_v, f_v)$ 和 (t_m, f_m)。如果 $t_v > t_m$，则新的界定点是 $(t_m, f_m), (t_v, f_v)$ 和 (t_1, f_1)。如果 $t_v = t_m$，则新的界定点不能用简单的方法来确定。而且，在该处终止迭代的条件并不充分，只是构建了一个例子，即三个样本点形成了一个等边三角形，其位于对称轴上的顶点就是抛物线的顶点，但是其全局最小值却远离该顶点。一种简单的启发式的方法，就是利用其中一个半区间的中点，比如说，$t_b = (t_0 + t_m)/2$，计算 $f_b = f(t_b)$，并与 f_m 进行比较。如果 $f_b > f_m$，那么新的界定点是 (t_b, f_b)，(t_m, f_m) 和 (t_1, f_1)。如果 $f_b < f_m$，则新的界定点为 $(t_0, f_0), (t_b, f_b)$ 和 (t_m, f_m)。如果 $f_b = f_m$，则另一个半区间可被对分，并重复进行相同的测试处理。如果这样也产生病态等式的情形，则可试试从 $[t_0, t_1]$ 内进行随机采样。一旦知道新的界定点，则重

复该方法，直到满足一些停止条件。

可以修改 Brent 法以支持导数信息。Press 等（1988）中也提供了这类描述。

A.6.2 多维方法

考虑 $f: D \subset \mathbb{R}^n \to \mathbb{R}$，其中$D$ 是一个紧凑集。在一般的图形应用中，D 是多面体或者甚至是区间的笛卡尔乘积。如果f 是可微的，那么全局最小值必定出现在一个满足 $\vec{\nabla} f = \vec{0}$ 的点或者位于D的边界上。在后一种情形中，如果D 是一个多面体，那么对f 对D 的每一个面的限制都得到维数较小的相同类型的最小值问题。例如，在讨论几何方法的第 10 章中描述的许多距离方法都属于这类情形。求解$\vec{\nabla} f = \vec{0}$ 是一种求根的问题，这本身就是一种难以解决的问题。

1. 最速下降搜索

这是搜索可微函数最小值的最简单的方法。根据微积分可知，函数f 递减速率最大的方向是 $-\vec{\nabla} f$。给定出现最小值的点的一个初始估值 $\vec{x}$，可用一个一维函数来使函数 $\phi(t) = f(\vec{x} - t\vec{\nabla} f(\vec{x}))$ 最小化。如果t'是出现最小值的参数，那么 $\vec{x} \leftarrow \vec{x} - t'\vec{\nabla} f(\vec{x})$，并且重复算法，直到满足一个停止条件。这种条件一般是说明最后一个开始位置与最新求得的位置之间的差别的一种量度。

这种方法的问题是其速度可能很慢。抛物面$f(x, y) = (x/a)^2 + y^2$（其中a是一个非常大的数）的最小化问题是其中的一个病态情形。其水平曲线集是在x轴方向拉伸很长的椭圆。对于不在x轴上的点，梯度向量的负值趋近于几乎与y 轴平行。搜索路径将沿锯齿形地反复穿过x轴，这将花费大量的时间才能搜索到出现最小值的原点。一种更好的方法是不使用梯度向量，而使用所谓的共轭方向。对于上述的抛物面，不论初始估值是什么，只需进行两次共轭方向迭代，就总能结束于原点。这些方向在某种意义上来说概括了函数的水平曲线的形状信息。

2. 共轭梯度搜索

这种方法试图选取一种比最速下降搜索更好的搜索最小值的方向集。Press 等（1988）讨论了这种方法的主要思想，这里仅进行简单介绍。构建两种方向序列，一种是梯度方向序列$\vec{g}_i$，另一种是共轭方向序列$\vec{h}_i$。一维的最小值沿着对应于共轭方向的直线运动。下面的伪码使用了 Press 等（1988）中提到的 Polak 和 Ribiere 公式。进行最小化的函数是$E(\vec{x})$。函数`MinimizeOn`使用一个一维最小化方法沿着直线求函数的最小值。它返回出现最小值的位置`x`，以及函数的最小值`fval`。

```
x = initial guess;
g = -gradient(E)(x);
h = g;
while (not done) {
    line.origin = x;
    line.direction = h;
    MinimizeOn(line, x, fval);
    if (stopping condition met)
        return <x, fval>;
```

```
        gNext = -gradient(E)(x);
        c = Dot(gNext - g, gNext) / Dot(g, g);
        g = gNext;
        h = g + c * h;
    }
```

其停止条件可以是以 fval 的连续值为基础和 / 或以 x 的连续值为基础。Press 等（1988）中使用的条件是以函数值 f_0 和 f_1 的连续值和一个小的公差值 $\tau > 0$ 为基础的：

$$2|f_1 - f_0| \leqslant \tau(|f_0| + |f_1| + \epsilon)$$

其中 $\epsilon > 0$ 是当函数最小值为零时支持这种情形的很小的值。

3. 鲍威尔方向集方法

如果 f 是连续但不可微的函数，那么它在 D 上取得最小值。最小值的搜索不能简单地利用导数信息。一种不需要求导的寻找最小值的方法是鲍威尔方向集方法（Powell's direction set method）。这种方法沿着定义域内的直线路径来求解最小值问题。当前的可能出现最小值的点更新为位于当前考虑的直线上的最小值点。选取经过当前点并且方向为取自维护的方向向量集中的一个方向的直线为下一条直线。一旦处理完了所有的方向，就计算出一个新的方向集。该方向集一般以前一个方向集为基础，删除原来的第一个方向，并将当前位置减去直线的最小值处理之前的老位置所得到的向量设为方向集中的新方向。进行直线的最小值处理时，使用某种类似于 Brent 方法的算法，这样做的原因是限制为直线的函数 f 是一维函数。由于 D 是紧凑集，因此能保证直线与 D 的交集也是一个紧凑集。而且，如果 D 是凸的（大多数的应用都是如此），那么相交是一个连通的区间，因而可在该区间上使用 Brent 方法（而不是对与 D 的相交的每一个连通的分量应用这种方法）。鲍威尔方法的伪码如下：

```
// F(x) is the function to be minimized
n = dimension of domain;
directionSet = {d[0],..., d[n - 1]};  // usually the standard axis directions
x = xInitial = initial guess for minimum point;
while (not done) {
    for (each direction d) {
        line.origin = x;
        line.direction = d;
        MinimizeOn(line, x, fval);
    }
    conjugateDirection = x - xInitial;
    if (Length(conjugateDirection) is small)
        return <x, fval>;  // minimum found

    for (i = 0; i <= n - 2; i++)
        d[i] = d[i + 1];
    d[n - 1] = conjugateDirection;
}
```

函数 MinimizeOn 就是在讨论共轭梯度搜索的上一节中提到的函数。

A.6.3　二次形式的最小化

设A是$n\times n$的对称矩阵。函数$Q:\mathbb{R}^n\to\mathbb{R}$由$Q(\hat{v})=\hat{v}^{\mathrm{T}}\mathbf{A}\hat{v}$（$\|\hat{v}\|=1$）定义，叫做二次形式（quadratic form）。由于Q定义在$\mathbb{R}^n$内的单位球上，因此这是一个紧凑集，并且由于Q是连续的，因此在这个集合上它必定具有一个最大值和一个最小值。

设$\hat{v}=\sum_{i=1}^{n}c_i v_i$，其中$\mathbf{A}\hat{v}_i=\lambda_i\hat{v}_i,\lambda_1\leqslant\cdots\leqslant\lambda_n$和$\sum_{i=1}^{n}c_i^2=1$。即$\lambda_i$是A的特征值，而$\hat{v}_i$是对应的特征向量。展开该二次形式，可得

$$Q(\hat{v})=\left(\sum_{i=1}^{n}c_i\hat{v}_i^{\mathrm{T}}\right)\mathbf{A}\left(\sum_{j=1}^{n}c_j\hat{v}_j\right)=\sum_{i=1}^{n}\sum_{j=1}^{n}c_ic_j\hat{v}_i^{\mathrm{T}}\mathbf{A}\hat{v}_j=\sum_{k=1}^{n}\lambda_k c_k^2$$

最右边的和是A的特征值的凸组合，因此其最小值为λ_n，并且当$c_1=1$和其他的$c_i=0$时取得该最小值。因此有$\min Q(\hat{v})=\lambda_1=Q(\hat{v}_1)$。

A.6.4　受约束二次形式的最小化

在一些应用中，可能想要找到二次形式定义在一个单位超球S^{n-1}上的最小值，但是限定于该超球与一个超平面$\hat{n}\cdot X=0$（$\hat{n}$是一些特殊的法线向量）的相交。设A是$n\times n$的对称矩阵。设$\hat{n}\in\mathbb{R}^n$是一个单位长度向量。设$\hat{n}^{\perp}$表示$\hat{n}$的正交补向量。用$Q(\hat{v})=\hat{v}^{\mathrm{T}}\mathbf{A}\hat{v}$来定义$Q:\{\hat{n}\}^{\perp}\to\mathbb{R}$，其中$\|\hat{v}\|=1$。现在$Q$是定义在$(n-1)$维空间$\hat{n}^{\perp}$内的单位球上，因此它必定有一个最大值和一个最小值。

设$\hat{v}_1$至$\hat{v}_{n-1}$为$\hat{n}^{\perp}$的一个正交基底。设$\hat{v}=\sum_{i=1}^{n-1}c_i\hat{v}_i$，其中$\sum_{i=1}^{n}c_i^2=1$。设$\mathbf{A}\hat{v}_i=\sum_{j=1}^{n-1}\alpha_{ji}\hat{v}_j+\alpha_{ni}\hat{n}$，其中$\alpha_{ji}=\hat{v}_j^{\mathrm{T}}\mathbf{A}\hat{v}_i$（$1\leqslant i\leqslant n-1$且$1\leqslant j\leqslant n-1$），以及$\alpha_{ni}=\hat{n}^{t}\mathbf{A}\hat{v}_i$（$1\leqslant i\leqslant n-1$）。展开该二次形式，可得

$$Q(\hat{v})=\left(\sum_{i=1}^{n-1}c_i\hat{v}_i^{t}\right)\mathbf{A}\left(\sum_{j=1}^{n-1}c_j\hat{v}_j\right)=\sum_{i=1}^{n-1}\sum_{j=1}^{n-1}c_ic_j\alpha_{ij}=\vec{c}^{\mathrm{T}}\bar{\mathbf{A}}\vec{c}=:P(\vec{c})$$

其中二次形式$P:\mathbb{R}^{n-1}\to\mathbb{R}$满足上一节中说明的最小值条件。因此，$\min Q(\hat{v})=\min P(\vec{c})$，当$\vec{c}$和$\lambda$满足$\bar{\mathbf{A}}\vec{c}=\lambda\vec{c}$且$\lambda$是$\bar{\mathbf{A}}$的最小特征值时取得该最小值。下面的计算可得到一个确定最小值的矩阵公式：

$$\sum_{j=1}^{n-1}\alpha_{ij}c_j=\lambda c_i$$

$$\sum_{j=1}^{n-1}c_j\hat{v}_i=\lambda c_i\hat{v}_i$$

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-1}\alpha_{ij}c_j\hat{v}_i=\lambda\sum_{i=1}^{n-1}c_i\hat{v}_i$$

$$\sum_{j=1}\left(\sum_{i=1}^{n-1}\alpha_{ij}\hat{v}_i\right)c_j=\lambda\hat{v}$$

$$\sum_{j=1}^{n-1}\left(A\hat{v}_j-\alpha_{nj}\hat{n}\right)c_j=\lambda\hat{v}$$

$$A\left(\sum_{j=1}^{n-1}c_j\hat{v}_j\right)-\left(\sum_{j=1}^{n-1}\alpha_{nj}c_j\right)\hat{n}=\lambda\hat{v}$$

$$A\hat{v}-\left(\hat{n}^tA\hat{v}\right)\hat{n}=\lambda\hat{v}$$

$$(\mathrm{I}-\hat{n}\hat{n}^t)A\hat{v}=\lambda\hat{v}$$

因此，$\min Q(\hat{v})=\lambda_1=Q(\hat{v}_1)$，其中 λ_1 是对应于 $(\mathrm{I}-\hat{n}\hat{n}^{\mathrm{T}})\mathrm{A}$ 的特征向量 $\hat{v}_1$ 的最小正特征值。注意，第 $n-1$ 个特征向量是 $\hat{n}^{\perp}$。剩余的特征向量为 $\hat{v}_n=\mathrm{A}^{\mathrm{adj}}\hat{n}$，其中 $\mathrm{AA}^{\mathrm{adj}}=(\det \mathrm{A})\mathrm{I}$ 且 $\lambda_n=0$。

A.7 最小二乘拟合

最小二乘拟合是选择一个用连续方式来表示一个离散点集的参数方程的一种处理方法。通过最小化这些参数的一个非负函数来估算这些参数。本节讨论直线拟合、平面拟合、圆拟合、球面拟合、二次曲线拟合和二次曲面拟合。

A.7.1 点的线性拟合

当表示测量点的数据的 y 分量是依赖于 x 分量的函数时，点的线性拟合一般是指用直线来进行最小二乘拟合。给定一个样本点的集合 $\{(x_i,y_i)\}_{i=1}^m$，确定 a 和 b 使得直线 $y=ax+b$ 最符合这些样本点，亦即使 y_i 和直线值 ax_i+b 之间的方差和最小。注意，仅仅测算 y 方向的误差。

定义 $E(a,b)=\sum_{i=1}^m[(ax_i+b)-y_i]^2$。该函数是非负的，并且其图形是一个抛物面，梯度满足 $\vec{\nabla}E=(0,0)$ 的位置就是其顶点。这样就得到一个易于求解的由两个关于 a 和 b 的线性方程所确定的系统。即，

$$(0,0)=\vec{\nabla}E=2\sum_{i=1}^m[(ax_i+b)-y_i](x_i,1)$$

因此有

$$\begin{bmatrix}\sum_{i=1}^m x_i^2 & \sum_{i=1}^m x_i\\ \sum_{i=1}^m x_i & \sum_{i=1}^m 1\end{bmatrix}\begin{bmatrix}a\\ b\end{bmatrix}=\begin{bmatrix}\sum_{i=1}^m x_iy_i\\ \sum_{i=1}^m y_i\end{bmatrix}$$

这种方法提供了最小二乘解 $y=ax+b$。

A.7.2 使用正交回归实现点的线性拟合

如果正交而不是垂直地测算与目标直线的误差，也可以使用最小二乘来拟合直线。在下面的讨论中，样本点和直线都位于 n 维空间中。设直线为 $L(t)=t\hat{d}+A$，其中 $\hat{d}$ 是单位

长度的。定义X_i为样本点，那么

$$X_i = A + d_i\hat{d} + p_i\hat{d}_i^{\perp}$$

其中$d_i = \hat{d} \cdot (X_i - A)$和$\hat{d}_i^{\perp}$是垂直于$\hat{d}$的单位长度向量，其相关系数为$p_i$。定义$\vec{y}_i = X_i - A$。从$X_i$到其在直线上的投影点的向量为

$$\vec{y}_i - d_i\hat{d} = p_i\hat{d}_i^{\perp}$$

该向量的长度平方为$p_i^2 = (\vec{y}_i - d_i\hat{d})^2$。最小二乘最小值的能量函数是$E(A,\hat{d}) = \sum_{i=1}^m p_i^2$。该函数的另外两种形式为

$$E(A,\hat{d}) = \sum_{i=1}^{m}\left(\vec{y}_i^t\left[\mathrm{I} - \hat{d}\hat{d}^t\right]\vec{y}_i\right)$$

和

$$E(A,\hat{d}) = \hat{d}^t\left(\sum_{i=1}^{m}\left[(\vec{y}_i \cdot \vec{y}_i)\mathrm{I} - \vec{y}_i\vec{y}_i^t\right]\right)\hat{d} = \hat{d}^t\mathbf{M}(A)\hat{d}$$

使用上一个方程中的E的第一种形式，求其对A的微分，可得

$$\frac{\partial E}{\partial A} = -2\left[\mathrm{I} - \hat{d}\hat{d}^t\right]\sum_{i=1}^{m}\vec{y}_i$$

当$\sum_{i=1}^m \vec{y}_i = 0$时，该偏微分为零，其中$A = (1/m)\sum_{i=1}^m X_i$为样本点的平均值。

给定A，矩阵$\mathbf{M}(A)$由上述能量方程的第二种形式来确定。数值$\hat{d}^t\mathbf{M}(A)\hat{d}$是一个二次形式，其最小值是$\mathbf{M}(A)$的最小特征值。这可用特征系统的标准求解方法来求解。再加上对应的单位长度特征向量$\hat{d}$，就完整地建构了最小二乘直线。

对于$n=2$，如果$A = (a, b)$，那么矩阵$\mathbf{M}(A)$由下式给定：

$$\mathbf{M}(A) = \left(\sum_{i=1}^{m}(x_i - a)^2 + \sum_{i=1}^{n}(y_i - b)^2\right)\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} - \begin{bmatrix}\sum_{i=1}^m(x_i - a)^2 & \sum_{i=1}^m(x_i - a)(y_i - b)\\ \sum_{i=1}^m(x_i - a)(y_i - b) & \sum_{i=1}^m(y_i - b)^2\end{bmatrix}$$

对于$n=3$，如果$A = (a, b, c)$，那么矩阵$\mathbf{M}(A)$由下式给定：

$$\mathbf{M}(A) = \delta\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix} - \begin{bmatrix}\sum_{i=1}^m(x_i - a)^2 & \sum_{i=1}^m(x_i - a)(y_i - b) & \sum_{i=1}^m(x_i - a)(z_i - c)\\ \sum_{i=1}^m(x_i - a)(y_i - b) & \sum_{i=1}^m(y_i - b)^2 & \sum_{i=1}^m(y_i - b)(z_i - c)\\ \sum_{i=1}^m(x_i - a)(z_i - c) & \sum_{i=1}^m(y_i - b)(z_i - c) & \sum_{i=1}^m(z_i - c)^2\end{bmatrix}$$

其中

$$\delta=\sum_{i=1}^{m}(x_i-a)^2+\sum_{i=1}^{m}(y_i-b)^2+\sum_{i=1}^{m}(z_i-c)^2$$

A.7.3 点的平面拟合

这里的假设前提是数据的z分量是依赖于x分量和y分量的函数。给定一个样本点的集合$\{(x_i,y_i,z_i)\}_{i=1}^{m}$，确定$a$，$b$和$c$，使得平面$z=ax+by+c$最符合样本点，亦即$z_i$与平面值$ax_i+by_i+c$之间的方差之和最小。注意，仅测算$z$方向上的误差。

定义$E(a,b,c)=\sum_{i=1}^{m}[(ax_i+by_i+c)-z_i]^2$。该函数是非负的，并且其图形是一个超抛物面，梯度满足$\vec{\nabla}E=(0,0,0)$之处就是其顶点。这样就得到一个易于求解的由三个关于a，b和c的线性方程所确定的系统。即，

$$(0,0,0)=\vec{\nabla}E=2\sum_{i=1}^{m}[(ax_i+by_i+c)-z_i](x_i,y_i,1)$$

因此有

$$\begin{bmatrix}\sum_{i=1}^{m}x_i^2 & \sum_{i=1}^{m}x_iy_i & \sum_{i=1}^{m}x_i\\ \sum_{i=1}^{m}x_iy_i & \sum_{i=1}^{m}y_i^2 & \sum_{i=1}^{m}y_i\\ \sum_{i=1}^{m}x_i & \sum_{i=1}^{m}y_i & \sum_{i=1}^{m}1\end{bmatrix}\begin{bmatrix}a\\ b\\ c\end{bmatrix}=\begin{bmatrix}\sum_{i=1}^{m}x_iz_i\\ \sum_{i=1}^{m}y_iz_i\\ \sum_{i=1}^{m}z_i\end{bmatrix}$$

这种方法提供了最小二乘解$z=ax+by+c$。

A.7.4 使用正交回归实现点的超平面拟合

如果正交而不是垂直地测算与目标平面的误差，也可以使用最小二乘来拟合平面。在下面的讨论中，样本点和超平面都位于n维空间中。设超平面为$\hat{n}\cdot(X-A)=0$，其中$\hat{n}$是超平面的单位长度法线，并且A是超平面上的一个点。定义X_i为样本点，那么

$$X_i=A+\lambda_i\hat{n}+p_i\hat{n}_i^{\perp}$$

其中$\lambda_i=\hat{n}\cdot(X_i-A)$和$\hat{n}_i^{\perp}$是垂直于$\hat{n}$的单位长度向量，其相关系数为$p_i$。定义$\vec{y}_i=X_i-A$。从$X_i$到其在超平面上的投影点的向量为$\lambda_i\hat{n}$。该向量的长度平方为$\lambda_i^2=(\hat{n}\cdot\vec{y}_i)^2$。最小二乘最小值的能量函数是$E(A,\hat{n})=\sum_{i=1}^{m}\lambda_i^2$。该函数的另外两种形式为

$$E(A,\hat{n})=\sum_{i=1}^{m}\left(\vec{y}_i^t\left[\hat{n}\hat{n}^t\right]\vec{y}_i\right)$$

和

$$E(A,\hat{n})=\hat{n}^t\left(\sum_{i=1}^{m}\vec{y}_i\vec{y}_i^t\right)\hat{n}=\hat{n}^t\mathrm{M}(A)\hat{n}$$

使用上一个方程中的E的第一种形式，求其对A的微分，可得

$$\frac{\partial E}{\partial A}=-2\left[\hat{n}\hat{n}^t\right]\sum_{i=1}^{m}\vec{y}_i$$

当$\sum_{i=1}^{m}\vec{y}_i=0$时，该偏微分为零，其中$A=(1/m)\sum_{i=1}^{m}X_i$为样本点的平均值。

给定A，矩阵$\mathrm{M}(A)$由上述能量方程的第二种形式来确定。数值$\hat{n}^t\mathrm{M}(A)\hat{n}$是一个二次形式，其最小值是$\mathrm{M}(A)$的最小特征值。这可用特征系统的标准求解方法来求解。再加上对应的单位长度特征向量$\hat{n}$，就完整地构建了最小二乘超平面。

对于$n=3$，如果$A=(a,b,c)$，那么矩阵$\mathrm{M}(A)$由下式给定：

$$\mathrm{M}(A)=\begin{bmatrix}\sum_{i=1}^{m}(x_i-a)^2 & \sum_{i=1}^{m}(x_i-a)(y_i-b) & \sum_{i=1}^{m}(x_i-a)(z_i-c)\\ \sum_{i=1}^{m}(x_i-a)(y_i-b) & \sum_{i=1}^{m}(y_i-b)^2 & \sum_{i=1}^{m}(y_i-b)(z_i-c)\\ \sum_{i=1}^{m}(x_i-a)(z_i-c) & \sum_{i=1}^{m}(y_i-b)(z_i-c) & \sum_{i=1}^{m}(z_i-c)^2\end{bmatrix}$$

A.7.5 将圆拟合到二维点集

给定一个点集$\{(x_i,y_i)\}_{i=1}^{m}, m\geq 3$，用圆$(x-a)^2+(y-b)^2=r^2$来拟合它们，其中（$a$，$b$）是圆心，且$r$是圆的半径。这种算法的假设前提是所有的点不全位于同一条直线上。将要最小化的能量函数是

$$E(a,b,r)=\sum_{i=1}^{m}(L_i-r)^2$$

其中$L_i=\sqrt{(x_i-a)^2+(y_i-b)^2}$。求其对$r$的偏微分，可得

$$\frac{\partial E}{\partial r}=-2\sum_{i=1}^{m}(L_i-r)$$

设其等于零，可得

$$r=\frac{1}{m}\sum_{i=1}^{m}L_i$$

求其对a的偏微分，可得

$$\frac{\partial E}{\partial a}=2\sum_{i=1}^{m}(L_i-r)\frac{\partial L_i}{\partial a}=-2\sum_{i=1}^{m}\left((x_i-a)+r\frac{\partial L_i}{\partial a}\right)$$

并求其对b的偏微分，可得

$$\frac{\partial E}{\partial b}=2\sum_{i=1}^{m}(L_i-r)\frac{\partial L_i}{\partial b}=-2\sum_{i=1}^{m}\left((y_i-b)+r\frac{\partial L_i}{\partial b}\right)$$

设这两个偏微分等于零，可得

$$a=\frac{1}{m}\sum_{i=1}^{m}x_i+r\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L_i}{\partial a} \text{ 和 } b=\frac{1}{m}\sum_{i=1}^{m}y_i+r\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L_i}{\partial b}$$

用从$\partial E/\partial r=0$中求得的r值代入上面的式子，并利用$\partial L_i/\partial a=(a-x_i)/L_i$和$\partial L_i/\partial b=(b-y_i)/L_i$，得到两个关于$a$和$b$的非线性方程：

$$a=\bar{x}+\bar{L}\bar{L}_a=:F(a,b),\quad b=\bar{y}+\bar{L}\bar{L}_b=:G(a,b)$$

其中

$$\bar{x}=\frac{1}{m}\sum_{i=1}^{m}x_i,\quad \bar{y}=\frac{1}{m}\sum_{i=1}^{m}y_i,\quad \bar{L}=\frac{1}{m}\sum_{i=1}^{m}L_i,\quad \bar{L}_a=\frac{1}{m}\sum_{i=1}^{m}\frac{a-x_i}{L_i},$$

$$\bar{L}_b=\frac{1}{m}\sum_{i=1}^{m}\frac{b-y_i}{L_i}$$

可以使用顶点迭代来求解这些方程：$a_0=\bar{x}, b_0=\bar{y}$，$a_{i+1}=F(a_i,b_i)$及$b_{i+1}=G(a_i,b_i)$，其中$i\geqslant 0$。

A.7.6 将球面拟合到三维点集

给定一个点集$\{(x_i,y_i,z_i)\}_{i=1}^{m}$, $m\geqslant 4$，用球$(x-a)^2+(y-b)^2+(z-c)^2=r^2$来拟合它们，其中（$a$，$b$，$c$）是球心，且$r$是球的半径。这种算法的假设前提是所有的点不全位于同一个平面上。将要最小化的能量函数是

$$E(a,b,c,r)=\sum_{i=1}^{m}(L_i-r)^2$$

其中$L_i=\sqrt{(x_i-a)^2+(y_i-b)^2+(z_i-c)}$。求其对$r$的偏微分，可得

$$\frac{\partial E}{\partial r}=-2\sum_{i=1}^{m}(L_i-r)$$

设其等于零，可得

$$r=\frac{1}{m}\sum_{i=1}^{m}L_i$$

求其对a的偏微分，可得

$$\frac{\partial E}{\partial a}=2\sum_{i=1}^{m}(L_i-r)\frac{\partial L_i}{\partial a}=-2\sum_{i=1}^{m}\left((x_i-a)+r\frac{\partial L_i}{\partial a}\right)$$

求其对b的偏微分，可得

$$\frac{\partial E}{\partial b}=2\sum_{i=1}^{m}(L_i-r)\frac{\partial L_i}{\partial b}=-2\sum_{i=1}^{m}\left((y_i-b)+r\frac{\partial L_i}{\partial b}\right)$$

并求其对c的偏微分，可得

$$\frac{\partial E}{\partial c}=2\sum_{i=1}^{m}(L_i-r)\frac{\partial L_i}{\partial c}=-2\sum_{i=1}^{m}\left((z_i-c)+r\frac{\partial L_i}{\partial c}\right)$$

设这三个偏微分等于零，可得

$$a=\frac{1}{m}\sum_{i=1}^{m}x_i+r\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L_i}{\partial a},\quad b=\frac{1}{m}\sum_{i=1}^{m}y_i+r\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L_i}{\partial b}\text{ 和 }c=\frac{1}{m}\sum_{i=1}^{m}z_i+r\frac{1}{m}\sum_{i=1}^{m}\frac{\partial L_i}{\partial c}$$

用从$\partial E/\partial r=0$中求得的r值代入上面的式子，并利用$\partial L_i/\partial a=(a-x_i)/L_i$，$\partial L_i/\partial b=(b-y_i)/L_i$ 和 $\partial L_i/\partial c=(c-z_i)/L_i$ 得到三个关于a，b 和c的非线性方程：

$$a=\bar{x}+\bar{L}\bar{L}_a=:F(a,b,c),\quad b=\bar{y}+\bar{L}\bar{L}_b=:G(a,b,c),\quad c=\bar{z}+\bar{L}\bar{L}_c=:H(a,b,c)$$

其中

$$\bar{x}=\frac{1}{m}\sum_{i=1}^{m}x_i,\quad \bar{y}=\frac{1}{m}\sum_{i=1}^{m}y_i,\quad \bar{z}=\frac{1}{m}\sum_{i=1}^{m}z_i$$

$$\bar{L}=\frac{1}{m}\sum_{i=1}^{m}L_i,\quad \bar{L}_a=\frac{1}{m}\sum_{i=1}^{m}\frac{a-x_i}{L_i},\quad \bar{L}_b=\frac{1}{m}\sum_{i=1}^{m}\frac{b-y_i}{L_i},\quad \bar{L}_c=\frac{1}{m}\sum_{i=1}^{m}\frac{c-z_i}{L_i}$$

可以使用顶点迭代来求解这些方程：$a_0=\bar{x},b_0=\bar{y},c_0=\bar{z}$，$a_{i+1}=F(a_i,b_i,c_i),b_{i+1}=G(a_i,b_i,c_i)$，以及$c_{i+1}=H(a_i,b_i,c_i)$，其中$i\geq 0$。

A.7.7 将二次曲线拟合到二维点集

给定一个点集$\{(x_i,y_i)\}_{i=0}^{n}$，现要用一条形式为$Q(x,y)=c_0+c_1x+c_2y+c_3x^2+c_4y^2+c_5xy=0$的二次曲线来拟合这些点。给定值$c_i$来提供这种拟合，任何数量与它的乘积也都提供了相同的拟合。为了减少自由方次，要求$\hat{c}=(c_0,\ldots,c_5)$具有单位长度。定义向量变量$\vec{v}=(1,x,y,x^2,y^2,xy)$。该二次方程变换为$Q(\vec{v})=\hat{c}\cdot\vec{v}=0$，并为$\vec{v}$的空间上的线性方程。为第 i 个数据点定义$\vec{v}_i=(1,x_i,y_i,x_i^2,y_i^2,x_iy_i)$。由于$Q(\vec{v}_i)$一般不为零，其基本思想是最小化平方和

$$E(\hat{c})=\left(\sum_{i=0}^{n}\hat{c}\cdot\vec{v}_i\right)^2=\hat{c}^{\mathrm{T}}\mathbf{M}\hat{c}$$

其中$\mathbf{M}=\sum_{i=0}^{n}\vec{v}_i\vec{v}_i^{\mathrm{T}}$，并且满足限制条件$\|\hat{c}\|=1$。现在的问题转换为标准二次形式的最小化问题，A.6 节讨论了这类问题。其最小值是矩阵$\mathbf{M}$的最小特征值，而$\hat{c}$是其对应的单位长度特征向量。该最小值也可用来评测拟合是否合适（该最小值为零则说明拟合严格精确）。

如果可以确定输入点几乎都位于同一个圆上，则可对上述构建进行一点小的改进。圆的形式为$Q(x,y)=c_0+c_1x+c_2y+c_3(x^2+y^2)=0$。可以进行相同的构建，其中$\vec{v}=(1,x,y,x^2+y^2$ 和$E(\hat{c})=\hat{c}^{\mathrm{T}}\mathbf{M}\hat{c}$，都满足条件$|\hat{c}|=1$。

A.7.8 将二次曲面拟合到三维点集

给定一个点集$\{(x_i,y_i,z_i)\}_{i=0}^{n}$，现要用一个形式为$Q(x,y,z)=c_0+c_1x+c_2y+c_3z+c_4x^2+c_5y^2+c_6z^2+c_7xy+c_8xz+c_9yz=0$的二次曲面来拟合这些点。与前一节一样，$\hat{c}=(c_i)$要求是单位长度的，并且$\vec{v}=(1,x,y,z,x^2,y^2,z^2,xy,xz,yz)$。要最小化的二次形式是$E(\hat{c})=\hat{c}^{\mathrm{T}}\mathbf{M}\hat{c}$，其中$\mathbf{M}=\sum_{i=0}^{2}\vec{v}_i\vec{v}_i^{\mathrm{T}}$。其最小值是矩阵$\mathbf{M}$的最小特征值，而$\hat{c}$是其对应的单位长度特征向量。该最小值也可用来评测拟合是否合适（该最小值为零则说明拟合严格精确）。

如果可以确定输入点几乎都位于同一个球面上，则可对上述构建进行一点小的改进。圆的形式为$Q(x,y,z)=c_0+c_1x+c_2y+c_3z+c_4(x^2+y^2+z^2)=0$。可以进行相同的构建，

其中$\tilde{v}=(1,x,y,z,x^2+y^2+z^2)$和$E(\hat{c})=\hat{c}^{\mathrm{T}}\mathrm{M}\hat{c}$，都满足条件$\|\hat{c}\|=1$。

A.8 曲线剖分

有时我们希望用一条折线来近似表示一条曲线$X(t)$（$t\in[t_{\min},t_{\max}]$）。这里讨论的方法适用于任意维曲线。一条曲线的剖分（subdivision）是一个对应于参数$\{t_i\}_{i=0}^{n}\subset[t_{\min},t_{\max}]$的一种递增次序的点集，其中$n$为一个确定的值。一般地，要求曲线的端点属于剖分，此时有$t_0=t_{\min}$和$t_n=t_{\max}$。剖分点为$X_i=X(t_i)$，其中$0\leqslant i\leqslant n$。本节讨论了几种有用的剖分方法。

A.8.1 基于均匀采样的剖分

最简单的剖分方法是对$[t_{\min},t_{\max}]$进行均匀采样。参数值为$t_i=t_{\min}+(t_{\max}-t_{\min})i/n$，其中$0\leqslant i\leqslant n$。这类采样易于实现，但是它忽略了曲线的固有变化。结果是，这种剖分产生的折线并不总是有效的近似。这种剖分的伪码如下：

```
Input:  Curve X(t) with t in [tmin, tmax]
        n, the number of subdivision points is n + 1
Output: subdivision {P[0],...,P[n]}

void Subdivide(int n, Point P[n + 1])
{
    for (int i = 0; i <= n; i++) {
        float t = tmin + (tmax - tmin) * i / n;
        P[i] = X(t);
    }
}
```

A.8.2 基于弧长的剖分

这种方法选取在曲线上沿着曲线均匀地分布的一个点集。两个点之间的间隔为沿着曲线测量的距离。这样得到的距离叫做弧长（arc length）。例如，如果曲线是半圆$x^2+y^2=1$（$y\geqslant 0$），当在平面上测量时，两个端点（1，0）和（−1，0）之间的距离是 2 个单位。由于两个点都在半圆上，它们之间的距离为π单位，那么这就是半圆的弧长。点$X_i=(\cos(\pi i/n),\sin(\pi i/n))$（$0\leqslant i\leqslant n$）为半圆的剖分。在平面上测得的距离$\|X_{i+1}-X_i\|$随着$i$而变化。然而，相邻的两个点之间的弧长为$\pi/n$，这是一个常数。这些点是关于弧长均匀分布的。

设L为曲线的总长度。设$t\in[t_{\min},t_{\max}]$为曲线参数，并且设$s\in[0,L]$为弧长参数。可以通过寻找位于$s_i=Li/n$（$0\leqslant i\leqslant n$）的点来构建剖分。这里的技术问题是，如何去定对应于s_i的t_i。根据一个指定的s来计算t的过程叫做基于弧长的再参数化（reparameterization by arc length）。这可通过关于s和t的积分方程的数值倒置来实现。定义$\mathrm{Speed}(t)=\|\vec{X}'(t)\|$和$\mathrm{Length}(t)=\int_{t_{\min}}^{t}\|\vec{X}'(\tau)\|\,d\tau$。问题转化为求解$\mathrm{Length}(t)-s=0$，$s$为指定值，这是一个求根的问题。根据弧长的定义，其根必须是惟一的。应用牛顿法就能求解该问题（参见 A.5 节）。

计算Length(t)需要用到数值积分。Press 等（1988）提供了不同的数值积分方法。根据s来计算t的伪码如下

```
Input: The curve X(t), domain [tmin, tmax], and total length L are available
    globally.  The Length and Speed calls implicitly use these values.  The
    value of s must be in [0, L].
Output:  The value of t corresponding to s.

float GetParameterFromArcLength(float s)
{
    // Choose an initial guess based on relative location of s in [0,L].
    float ratio = s / L;
    float t = (1 - ratio) * tmin + ratio * tmax;

    for (int i = 0; i < imax; i++) {
        float diff = Length(t) - s;
        if (|diff| < epsilon)
            return t;

        t -= diff / Speed(t);
    }

    // Newton's method failed to converge.  Return your best guess.
    return t;
}
```

应用程序必须选取迭代的最大值`imax`和一个公差`epsilon`用来表示根接近于零的程度。这种剖分的伪码为

```
Input: The curve X(t), domain [tmin, tmax], and total length L are available
    globally. The number of subdivision points is n + 1.
Output: subdivision {P[0],..., P[n]}

void Subdivide (int n, Point P[n + 1])
{
    for (int i = 0; i <= n; i++) {
        float s = L * i / n;
        float t = GetParameterFromArcLength(s);
        P[i] = X(t);
    }
}
```

A.8.3　基于中点距离的剖分

这种方法通过递归地对分参数空间以产生一种不均匀的采样。需要确切地执行对分，并要分析对应于中点参数所求得的曲线点。如果A和B是线段的端点，并且C是在对分步骤中计算得到的点，那么计算从C到线段的距离d_0。如果$d_1 = \|B - A\|$，那么B被增加到嵌面中，对于应用指定的最大相对误差$\varepsilon > 0$，有$d_0/d_1 > \varepsilon$。下面列出了这种方法的伪码。该伪码仅仅维护了一个关于有序点的单向链表来处理剖分中的点的插入，而不是维护一个双向链表。

```
Input: The curve X(t) and domain [tmin, tmax] are available globally.
       m, the maximum level of subdivision
       epsilon, the maximum relative error
       sub {}, an empty list
Output: n >= 1 and subdivision {p[0],..., p[n]}

void Subdivide (int level, float t0, Point X0, float t1, Point X1, List sub)
{
    if (level > 0) {
        tm = (t0 + t1) / 2;
        Xm = X(tm);
        d0 = length of segment <X0, X1>
        d1 = distance from Xm to segment <X0, X1>;

        if (d1 / d0 > epsilon) {
            Subdivide(level - 1, t0, X0, tm, Xm, sub);
            Subdivide(level - 1, tm, Xm, t1, X1, sub);
            return;
        }
    }

    sub.Append(X1);
}

Initial call:
    List sub = { X(tmin) };
    Subdivide(maxlevel, tmin, X(tmin), tmax, X(tmax), sub);
```

对d_0和d_1的计算需要进行费时的平方根运算。除法d_0/d_1也是很费时的。实现可能使用平方距离d_0^2和d_1^2，并进行比较$d_1^2 > \varepsilon^2 d_0^2$，以减小计算代价。

A.8.4 基于变分的剖分

剖分也可被看成是曲线的变换的反射。用均匀采样实现的直观的剖分不能获得曲线的几乎为线性的变换或者急剧的变换信息。如果目的是得到非常精确的测量，那么一种更好的方法是递归地剖分相应的曲线段，在这些位置上基于曲线和当前处理的线段之间的变分来使用剖分步骤。当所有变分都小于预先定义的公差时，递归终止。

变分尺度可以是多种数量中的一种。这里描述的是一种以曲线和线段之间的平方距离的积分为基础的。设$[t_0, t_1]$为当前处理的子区间。在该区间内的曲线端点为$X_i = X(t_i)$，其中$i = 0, 1$。用于近似的直线段为

$$\mathcal{L}(t) = X_0 + \frac{t - t_0}{t_1 - t_0}\left(X_1 - X_0\right)$$

用积分来定义的变分为

$$V([t_0, t_1]) = \int_{t_0}^{t_1} \|X(t) - \mathcal{L}(t)\|^2 dt$$

曲线与直线段之间的差别是一个多项式

$$X(t)-\mathcal{L}(t)=\sum_{i=0}^{n}B_i t^i$$

$B_0=A_0-(t_1X_0-t_0X_1)/(t_1-t_0)$, $B_1=A_1-(X_1-X_0)/(t_1-t_0)$ 且 $B_i=A_i$，其中 $2\leqslant i\leqslant n$。平方长度是一个次数为 $2n$ 的多项式：

$$\|X(t)-\mathcal{L}(t)\|^2=\sum_{i=0}^{n}B_i t^i\sum_{j=0}^{n}B_j t^j=\sum_{k=0}^{2n}\left(\sum_{m=\max\{0,k-n\}}^{k}B_{k-m}\cdot B_m\right)t^k$$

因此，变分为

$$V([t_0,t_1])=\sum_{k=0}^{2n}\left(\sum_{m=\max\{0,k-n\}}^{k}B_{k-m}\cdot B_m\right)\frac{t_1^{k+1}-t_0^{k+1}}{k+1}$$

这种剖分的伪码为

```
Input: The curve X(t) and domain [tmin, tmax] are available globally.
       maxlevel, the maximum level of subdivision;
       minvariation, the variation tolerance;
       sub {}, an empty list;
Output: n >= 1 and subdivision {p[0],..., p[n]};

void Subdivide(int level, float t0, Point X0, float t1, Point X1, List sub)
{
    if (level > 0) {
        var = Variation(t0, X0, t1, X1);
        if (var > minvariation) {
            tm = (t0 + t1) / 2;
            xmid = X(tm);
            Subdivide(level - 1, t0, X0, tm, Xm, sub);
            Subdivide(level - 1, tm, Xm, t1, X1, sub);
            return;
        }
    }

    sub.Append(X1);
}

Initial call:
    List sub = { X(tmin) };
    Subdivide(maxlevel, tmin, X(tmin), tmax, X(tmax), sub);
```

当一个小的minvariation被指定时，参数maxlevel用于避免深度递归调用。然而，如果level达到零值，当前区间的变分并不需要位于指定的公差之内。由于变分不提供距离计算的误差范围，这并不是问题。

A.9 与微积分相关的话题

本节复习了多个微积分话题，它们对于求解本书讨论的许多问题，特别是距离和相交问题，都非常有用。

A.9.1 水平集

政治地图显示地区和国家的边界，以及一些地理标志（河流、海洋等），但是它们通常都只是被示意性地显示。另一方面，地形图是非常有用的，因为它们明确地显示了陆地的高度。地形图通过等高线来表示高度，等高线就是表示陆地在特定高度的横截面的曲线。通过观察与特定等高线相关的高度表示，以及它们的地形状况，地图的使用者能方便地区分斜坡，山顶和山谷。而且，通过相邻的等高线之间的相对距离还能简单地知道斜坡的坡度，它们越接近则坡度越陡。

等高线地图可以看成是两个变量的函数 $f(x, y) = z$（其中 x 和 y 是精度和维度，z 是高度）的一种图示方法。如果将函数 f 与水平面

$$z = h$$

相交，并将所得的曲线投影到 xy 平面，那么我们就得到所谓的水平集（如图 A.2 所示）。阶层曲线具有方程

$$f(x, y) = a$$

如果我们有抛物面（如图 A.3 所示）

$$f(x, y) = \frac{2x^2}{3} + y^2$$

那么阶层曲线具有如下的方程形式

$$\frac{2x^2}{3} + y^2 = h$$

对于 $h>0$，它们为椭圆；对于 $h=0$，只是一个点；对于 $h<0$，不存在曲线。图 A.4 中显示了一些投影的阶层曲线。

函数 $f(x, y)$ 的阶层曲线和该函数的梯度 $\nabla f(x, y)$ 之间有非常重要的几何关系。假定我们有任意点（当然位于 f 的定义域内）x_0, y_0，并对某些常数 c 满足 $f(x_0, y_0) = c$。我们知道 $f(x, y) = c$ 定义了 f 的一些阶层曲线，并且 $f(x_0, y_0)$ 是曲线上的一个点。f 在 (x_0, y_0) 处具有最大递增速率的方向是 $\nabla f(x_0, y_0)$ 的方向，而具有最小递增速率的方向是 $-\nabla f(x_0, y_0)$ 的方向。从直观上这似乎说明，梯度必定在 $f(x_0, y_0)$ 处垂直于阶层曲线，并且实际上可以证明确实如此。下面给出了一种概要的证明（Anton 1980）。

假定我们具有阶层曲线

$$f(x, y) = c \tag{A.9}$$

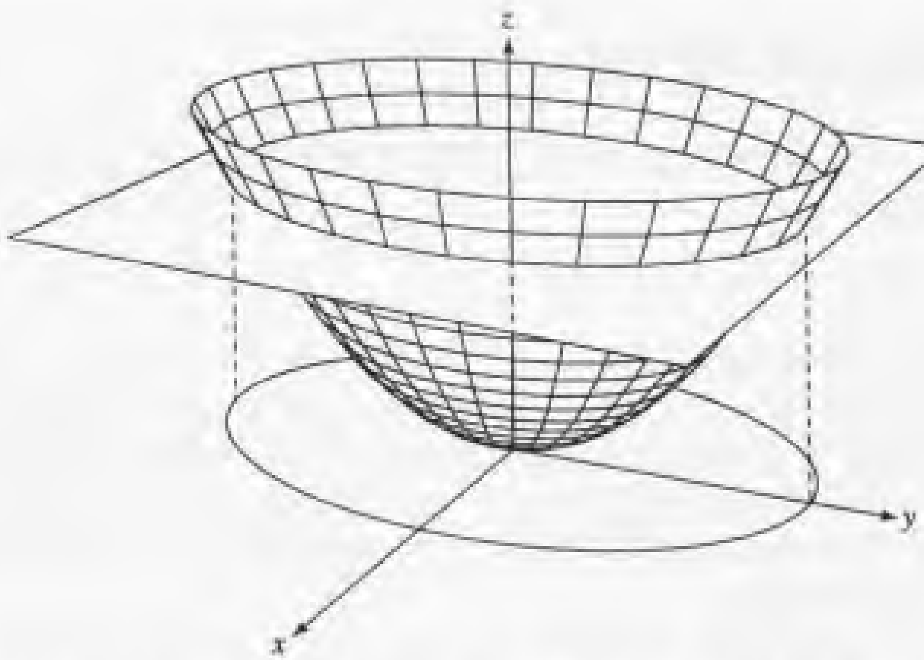


图 A.2　函数 $f(x, y) = z$ 和平面 $z = 0.8$ 相交产生一条阶层曲线，显示为在 xy 平面上的投影

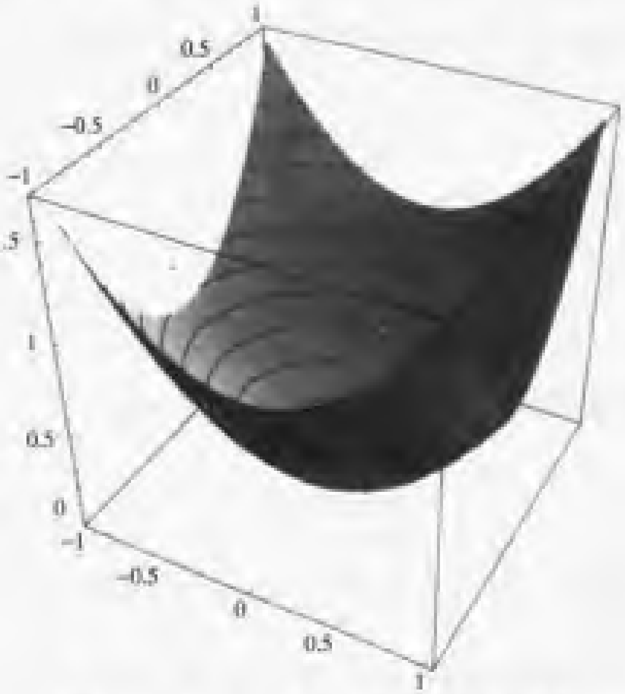


图 A.3　$f(x, y) = \frac{2x^2}{5} + y^2$ 表示的阶层曲线

它经过 (x_0, y_0)。我们可以用参数形式来将该曲线表示为

$$x = x(t)$$

$$y = y(t)$$

它在 (x_0, y_0) 处具有一个非零的切向量。

$$x'(t_0)\mathbf{i} + y'(t_0)\mathbf{j} \neq 0$$

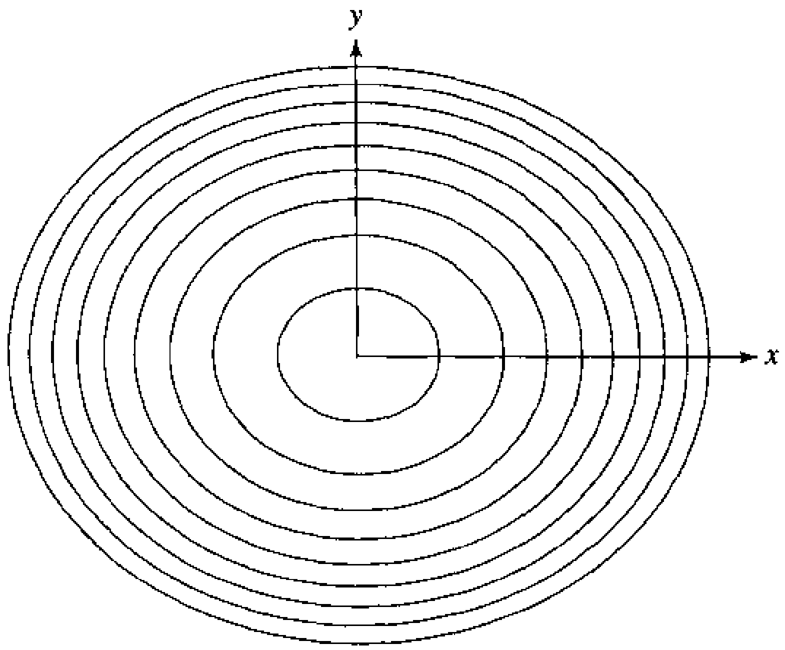


图 A.4 $f(x,y)=\frac{2x^2}{3}+y^2$ 表示的阶层曲线在 xy 平面上的投影

其中 t_0 是位于点 (x_0,y_0) 的参数值。

如果我们求方程（A.9）相对于 t 的微分，并应用微分链法则，可得

$$f_x(x(t),y(t))x'(t)+f_y(x(t),y(t))y'(t)=0$$

利用 $x(t_0)=x_0$ 和 $y(t_0)=y_0$，并代入 $t=t_0$，可得

$$f_x(x_0,y_0)x'(t_0)+f_y(x_0,y_0)y'(t_0)=0$$

该式可改写为

$$\nabla f(x_0,y_0)\cdot(x'(t_0)\mathbf{i}+y'(t_0)\mathbf{j})=0$$

就是 $\nabla f(x_0,y_0)$ 在 (x_0,y_0) 处垂直于阶层曲线的切向量。

A.9.2 函数的最大值和最小值

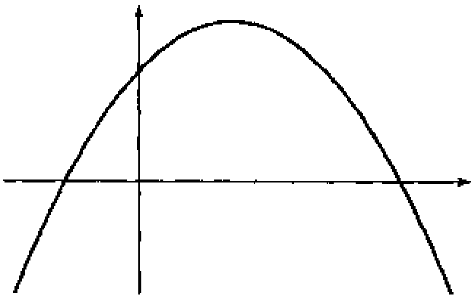
本节介绍寻找具有一个和两个变量的函数的最大值和最小值的问题。在所谓的优化问题中，规范地说，在寻找最优化方法时，我们经常遇到这种问题。

1. 一个变量的函数

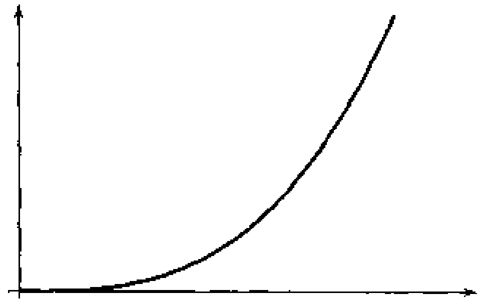
给定一个函数 f，我们可能需要知道关于最小值或者最大值的不同事实，比如

- $f(x)$ 是否具有最小值（最大值）？
- 如果 $f(x)$ 具有最小值（最大值），该值是多少，何时出现该值？
- $f(x)$ 是否在区间 $[a,b]$ 或者 $[a,b)$ 内具有最小值（最大值）？

图 A.5 显示了两个函数，其中一个在其定义域内没有最小值，而另一个则没有最大值。我们从几个相关的定义开始讨论。



无最小值



无最大值

图 A.5　无最小值或者最大值的两个函数

【定义 A.1】如果满足下面的条件，则数值 M 称为函数 f 在定义域 D 内的绝对最大值（absolute maximum value）：

i．$f(x) \leqslant M, \forall x \in D$.

ii．$\exists x \in D | f(x) = M$. ■

【定义 A.2】如果满足下面的条件，则数值 M 称为函数 f 在定义域 D 内的绝对最小值（absolute minimum value）：

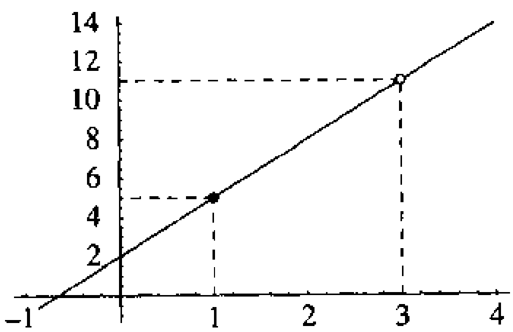
i．$f(x) \geqslant M, \forall x \in D$.

ii．$\exists x \in D | f(x) = M$. ■

我们常常对位于 f 的定义域内的某些受限部分内的极值感兴趣。

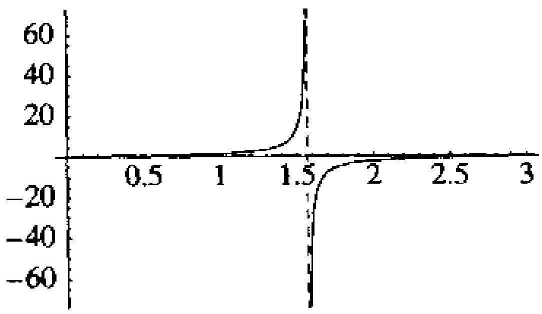
【定理 A.1】极值定理——一个变量的形式：设 $f(x)$ 在闭区间 $[a, b]$ 上是连续的，那么存在一些数值 $x_0, x_1 \in [a, b]$，它们满足 $f(x) \leqslant f(x_0)$ 和 $f(x) \geqslant f(x_1), \forall x \in [a, b]$。■

注意其中的两个条件，即 f 是连续的且区间是闭区间，都是必需的，这样才能保证存在相对极值。下面的两个例子中的每一个都分别违背了其中的一个条件，一个是 $\tan(x)$，$\forall x \in [0, 3]$，另一个是 $f(x) = 3x + 2$，$\forall x \in [2, 6)$。正切函数不满足条件是因为在值 1.5 处不连续，而直线函数不满足条件是因为其区间的右边不是闭合的（不存在"小于 11 的最大值"）。图 A.6 显示了它们的图形。



$f(x) = 3x + 2, 1 \leqslant x < 3$

右边不闭合



$f(x) = \tan x, 0 \leqslant x < 3$

不连续

图 A.6　违背极值定理假设条件的两个函数

极值定义仅仅定义了极值存在的条件，但是对于寻找极值却没有任何直接的帮助。为了求得极值，我们需要借助于临界点的概念。

【定义 A.3】 对于函数f，临界点（critical point）是任何满足如下条件之一的点$x \in D$：

i. 第一阶导数为$0 - f'(x) = 0$

ii. 第一阶导数不存在

满足$f'(x) = 0$的临界点叫做平稳点。■

图 A.7 显示了多种不同的临界点。图 A.7（a），图 A.7（b）和图 A.7（c）中的是平稳点，图 A.7（c）和图 A.7（d）中的叫做拐点（inflection point）。临界点非常重要，因为它们是确定极值的关键，下面的定理将说明这一点。

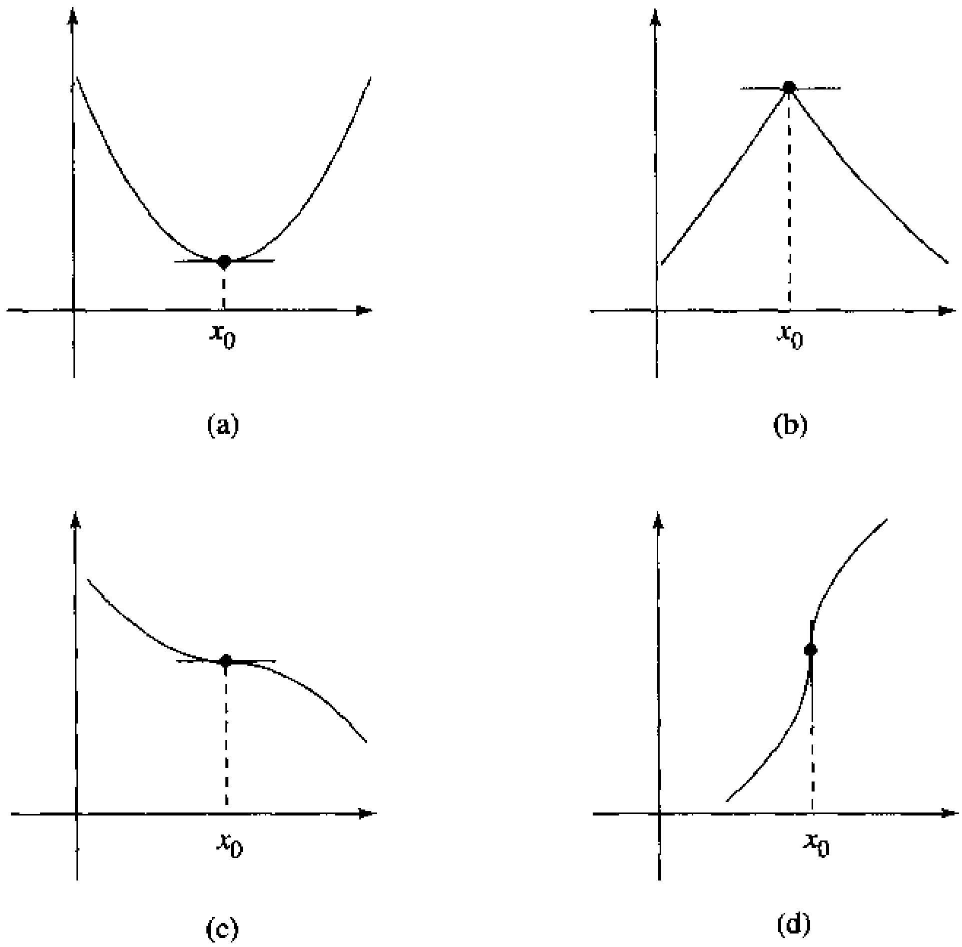


图 A.7 显示临界点的多个不同的函数。(a) 和 (b) 是平稳点；(c) 和 (d) 是拐点

【定理 A.2】 如果一个函数f在区间(a,b)上具有极值，那么极值出现在f的一个临界上。■

在 Anton（1980）中可以找到该定理的一种证明。这个定理也可以用来确定f在一个闭区间上的极值：极值或者出现在区间内，或者出现在区间的边界（端点）上。图 A.8 显示了函数的最大值出现在区间的右端点、区间内不可微的一点和区间内可微的一点的情形。因此，寻找一个（连续）函数在一个闭区间$[a,b]$上的极值的有效过程如下：

（1）寻找开区间(a,b)内的临界点。

（2）计算在区间边界 a 和 b 处的函数值。

（3）比较在临界点和区间端点的函数值。其中最小的就是最小值，最大的就是最大值。

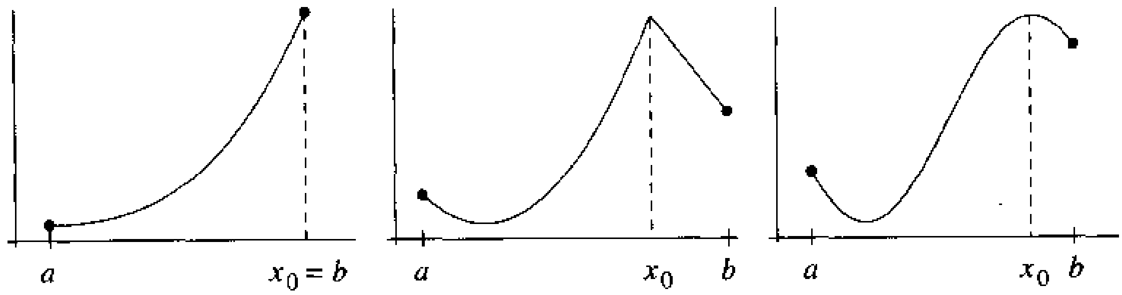


图 A.8　函数的最大值可能出现在区间的边界处或区间内部

下面的例子可以说明这一方法。考虑方程

$$f(x) = x^3 + 6x^2 - 7x + 19 \tag{A.10}$$

我们希望找到它在区间［−8，3］上的极值（如图 A.9 所示）。首先，我们必须校验该函数是连续的，这样才能应用极值定理。由于该函数是一个多项式，因此我们知道它是连续的。极值定理告诉我们，如果在开区间内存在极值，则极值必定出现在临界点上，而临界点（在这种情形中）必定出现在导数为零的点上，因此我们需要求 $f'(x)$ 以找到这些点。

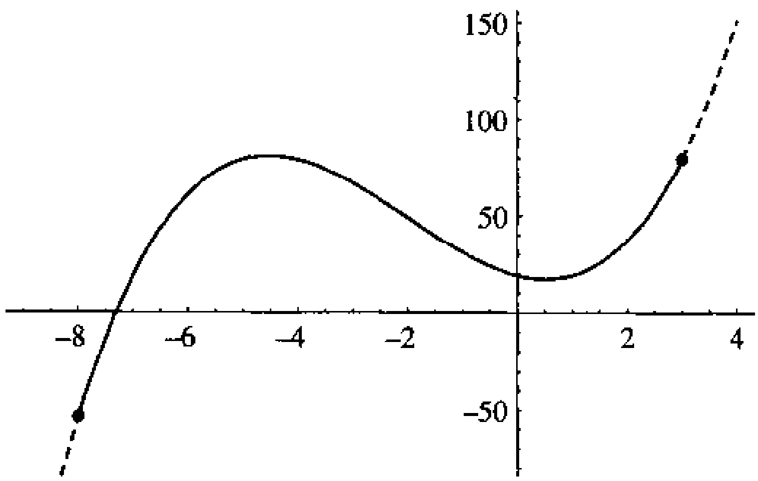


图 A.9　$f(x) = x^3 + 6x^2 - 7x + 19, \forall x \in [-8, 3]$

函数 f 的导数为

$$f'(x) = 3x^2 + 12x - 7 \tag{A.11}$$

其根为 $\{\{x \to \frac{-6-\sqrt{57}}{3}\}, \{x \to \frac{-6+\sqrt{57}}{3}\}\} \approx \{-4.51661, 0.516611\}$（参见图 A.10）。函数 f 在端点的值为 $f(-8)=-53$，$f(3)=79$。因此，最小值出现在左边界 $f(-8)=-53$，而最大值出现在临界点 $f(\frac{-6-\sqrt{57}}{3}) \approx f(-4.51661) \approx 80.8771$。

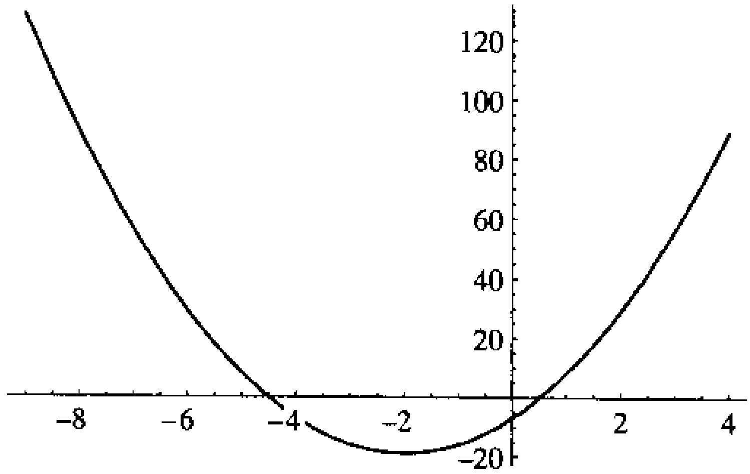


图 A.10 $f'(x) = 3x^2 + 12x - 7, \forall x \in [-8, 3]$

相对极值

只有当区间为闭区间并且函数是连续函数时才适用极值定理。因此，有一些情形不适合用极值定理。许多函数的图形含有一些区间，其中包含一般称为“峰”和“谷”的部分，称为相对极值（relative extrema）。图 A.11 显示了一个这样的例子。它们的正式定义如下：

【定义 A.4】 如果满足下面的条件，则函数 f 在 x_0 具有相对极大值：

$$\exists(a,b) \mid f(x_0) \geqslant \forall x \in (a,b) \ \blacksquare$$

【定义 A.5】 如果满足下面的条件，则函数 f 在 x_0 具有相对极小值

$$\exists(a,b) \mid f(x_0) \leqslant f(x), \forall x \in (a,b) \ \blacksquare$$

使用下面的三个定理可以确定相对极值。

【定理 A.3】 相对极值定理：如果函数 f 在点 x_0 具有相对极值，那么 x_0 是函数 f 的一个临界点。■

注意，该定理反过来并不成立，即一个临界点并不一定是一个极值点。图 A.7（c）中有一个临界点，但不是一个相对极值点。

【定理 A.4】 一阶导数检测：如果函数 f 在一个临界点 x_0 连续，那么

i．如果在一个右端点为 x_0 的开区间内有 $f'(x) > 0$，并且在一个左端点为 x_0 的开区间内有 $f'(x) < 0$，那么 $f(x_0)$ 是一个相对极大值。

ii．如果在一个右端点为 x_0 的开区间内有 $f'(x) < 0$，并且在一个左端点为 x_0 的开区间内有 $f'(x) > 0$，那么 $f(x_0)$ 是一个相对极小值。

iii．如果 $f'(x)$ 在以 x_0 为左端点的开区间和以该点为右端点的开区间内的符号相同，那么 $f(x_0)$ 不是一个相对极值。■

不严格地说，这类函数的相对极值出现在导数值改变符号的临界点处。

Anton（1988）中提供了一个例子，显示了如何利用该定理来寻找如下函数的相对极值：

$$f(x) = 3x^{\frac{5}{3}} - 15x^{\frac{2}{3}}$$

我们对 f 求导，以计算临界点：

$$f'(x) = 5x^{\frac{2}{3}} - 10x^{-\frac{1}{3}}$$

$$= 5x^{-\frac{1}{3}}(x-2)$$

那么临界点为 $x=0$（该点的导数不存在），以及 $x=2$（参见图 A.12）。

第三个定理如下：

【定理 A.5】 二阶导数检测：如果 f 在一个平稳点 x_0 处是二阶可导的，那么

i. 如果 $f''(x_0) > 0$，那么 $f(x_0)$ 是一个相对极小值。

ii. 如果 $f''(x_0) < 0$，那么 $f(x_0)$ 是一个相对极大值。■

该定理的直观解释如下：在一个二阶导数为负的临界点上，函数的图形是凸的；在一个二阶导数为正的临界点上，函数的图形是凹的。

下面的例子说明了如何使用这种方法。我们要寻找如下函数的局部极值：

$$f(x) = x^4 - 3x^2 + 3$$

我们求 f 的一阶导数，以计算临界点：

$$f'(x) = 4x^3 - 6x$$

即为

$$\{\{x \to -1.22474\}, \{x \to 0\}, \{x \to 1.22474\}\}$$

二阶导数为

$$f''(x) = 12x^2 - 6$$

如果我们将 f' 的平稳点代入 f''，可得

$$f''(-1.22474) = 12 > 0$$

$$f''(0) = -6 < 0$$

$$f''(1.22474) = 12 > 0$$

因此，我们可得出如下结论：函数在 $x=0$ 存在相对极大值，在 $x \approx -1.22474$ 和 $x \approx 1.22474$ 存在相对极小值。

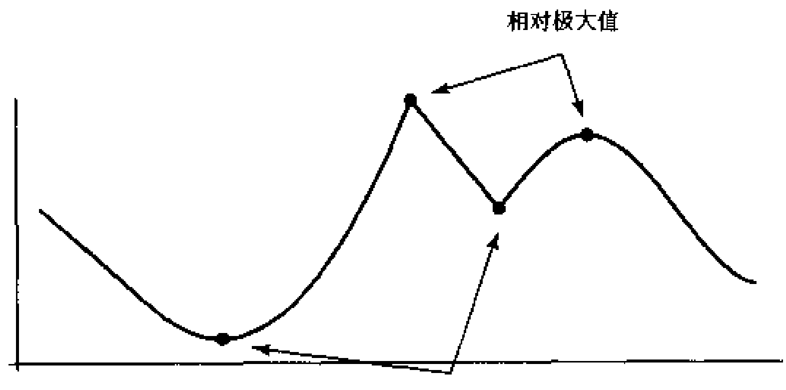


图 A.11　相对极值

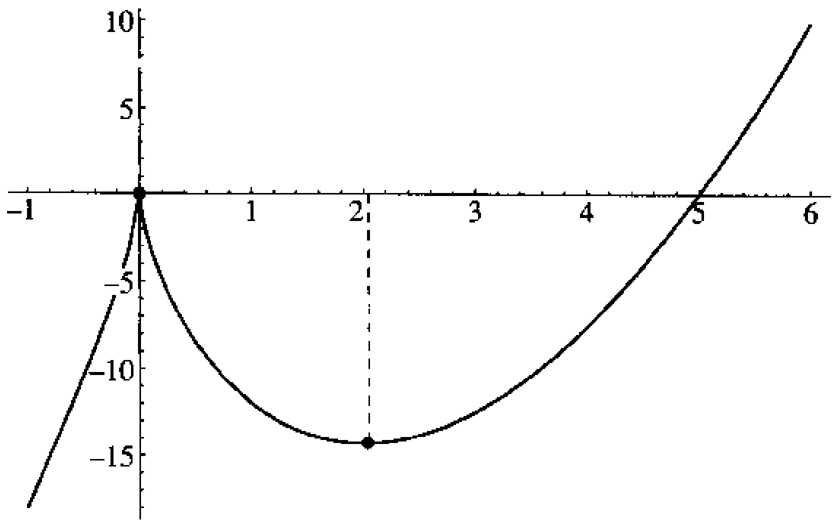


图 A.12 $f(x)=3x^{\frac{5}{3}}-15x^{\frac{2}{3}}$ 的相对极值

2. 两个以上变量的函数

前面我们介绍了寻找一个变量的函数的极值的方法。下面我们讨论寻找两个变量的函数的类似方法。可以很方便地用平面上的曲线来图示一个变量的函数，类似地，可以用三维空间中的曲面来图示两个变量的函数。由于曲面是曲线的三维相似体，因此我们也有一些优化技巧可以将这类用于曲线的方法扩展到曲面。

与一个变量的函数的情形一样，两个变量的函数的图形中的“峰和谷”也是相对极值（参见图 A.13）。其正式定义如下。

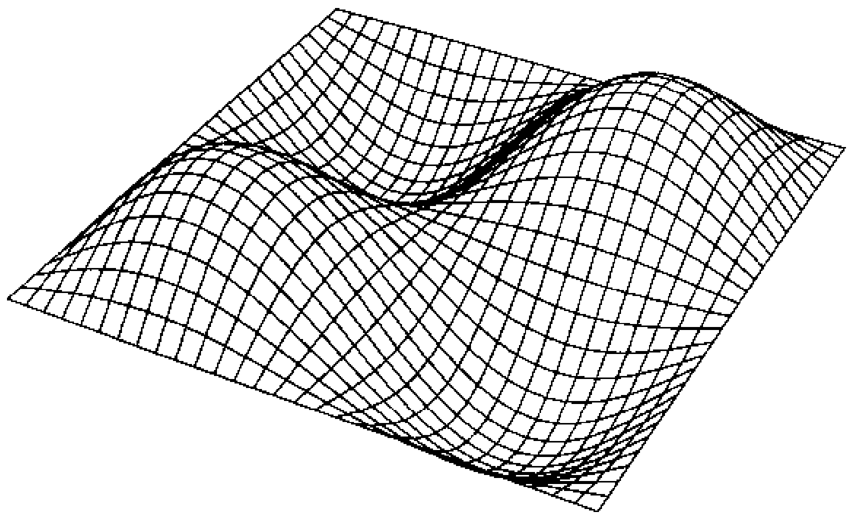


图 A.13 一个具有两个变量的函数的相对极值是其图形的峰和谷

【定义 A.6】如果存在一个圆心为(x_0, y_0)且满足$f(x_0, y_0)\geqslant f(x, y)$, $\forall x$位于圆内，那么两个变量的函数f在点(x_0, y_0)具有相对极大值。■

【定义 A.7】如果存在一个圆心为(x_0, y_0)且满足$f(x_0, y_0)\leqslant f(x, y)$, $\forall x$位于圆内，那么两个变量的函数f在点(x_0, y_0)具有相对极小值。■

其中的圆可以理解为含有单个变量的函数的定义域的线性区间的自然扩展。

绝对极值的定义如下。

【定义 A.8】 如果 $f(x_0, y_0) \geqslant f(x, y), \forall (x, y) \in D_1 \times D_2$，那么定义域为 D_1 和 D_2 的两个变量的函数 f 在点 (x_0, y_0) 具有绝对极大值。■

【定义 A.9】 如果 $f(x_0, y_0) \leqslant f(x, y), \forall (x, y) \in D_1 \times D_2$，那么定义域为 D_1 和 D_2 的含有两个变量的函数 f 在点 (x_0, y_0) 具有绝对极大值。■

对于一个变量的函数来说，其相对极值存在于一阶导数为零的点上。在图形上，这表现为出现水平的切线。对于两个变量的函数，类似的条件是，其极值存在于函数关于 x 和 y 的偏导数（假设都存在）都为零的点。在图形上这表现为，函数 $z = f(x, y)$ 在平面 $x = x_0$ 和 $y = y_0$ 上的迹线在 (x_0, y_0) 上都具有水平切线（如图 A.14 所示）。因此，我们有

$$\frac{\partial f}{\partial x}(x_0, y_0) = 0$$

和

$$\frac{\partial f}{\partial y}(x_0, y_0) = 0$$

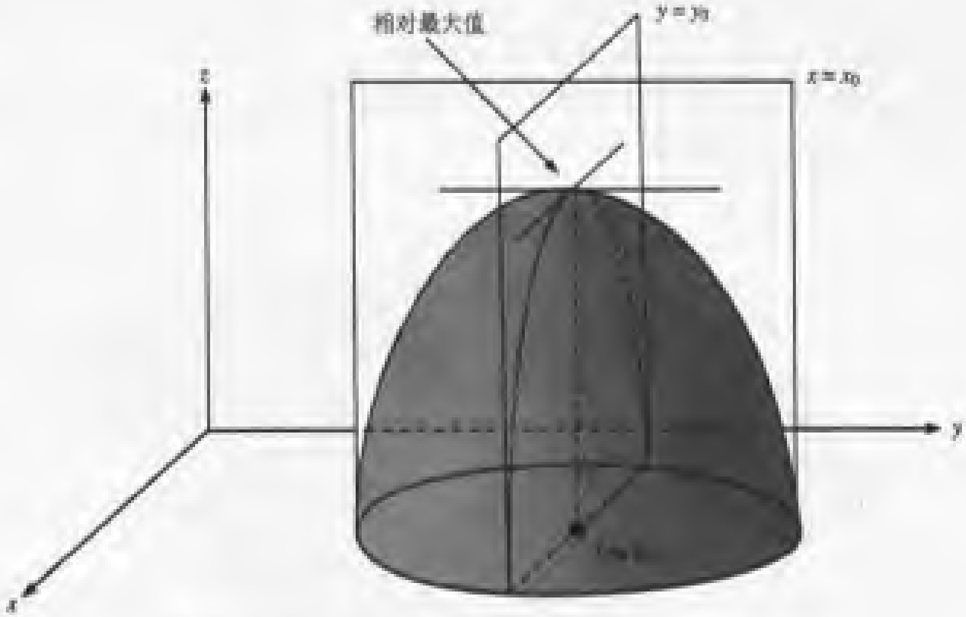


图 A.14　函数 $z = f(x, y)$ 的相对极值（Anton，1980）

下面的定理严格地描述了这一结果。

【定理 A.6】 如果两个变量的函数 f 在 (x_0, y_0) 具有一个相对极值，并且如果 f 的一阶偏导数在该点都存在，那么

$$\frac{\partial f}{\partial x}(x_0, y_0) = 0$$

和

$$\frac{\partial f}{\partial y}(x_0, y_0)=0 \quad ■$$

与单个变量的函数一样，函数$f(x,y)$的在定义域内的一阶偏导数都为零的点，叫做临界点。因此，上述定理说明了相对极值出现在临界点，与一个变量的函数的相关定理一致。

对于单个变量的函数，一阶导数为零的点并不一定就是相对极值点。函数图形上的拐点就是其中的这样的例子。类似地，对于两个变量的函数，偏导数都为零的点并不一定就是相对极值点。图 A.15 显示了函数$f(x,y)=x^2-y^2$的图形。在点（0，0），该函数在XZ和YZ平面上的迹线都具有水平切线，因为

$$\frac{\partial f}{\partial x}(x_0, y_0)=0$$

$$\frac{\partial f}{\partial y}(x_0, y_0)=0$$

注意，圆心为（0，0）的任何圆都既具有z值大于零的点也具有z值小于零的点，因此，（0，0）虽然是一个临界点，却不是一个极值点。依其形状，这类点叫做鞍点（saddle point）。

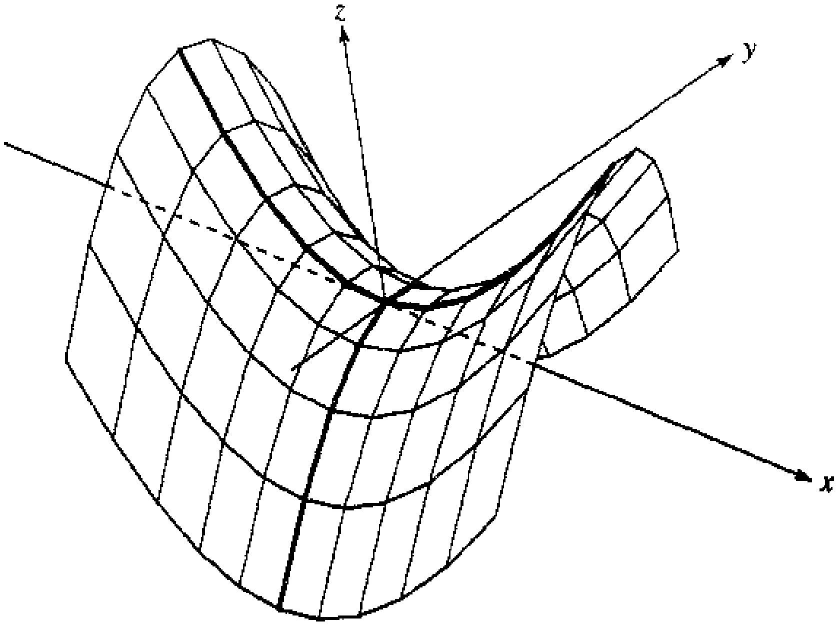


图 A.15 一个"鞍形函数"——点（0，0）不是一个极值点，虽然第一个偏导数为 0

我们可以利用一阶偏导数来确定函数的相对极值点和鞍点。例如，给定一个函数

$$f(x,y)=3-x^2-y^2$$

求其偏导数

$$\frac{\partial f}{\partial x}=-2x$$

$$\frac{\partial f}{\partial y}=-2y$$

并设它们等于零。这样可得$x=0$和$y=0$，因此（0，0）是惟一的临界点。计算函数在该

临界点的值，可得$f(0,0)=3$。对于所有除（0，0）之外的点(x,y)，我们有$f(x,y)<3$，因为$f(x,y)=3-x^2-y^2=3-(x^2+y^2)$。因此，（0，0）是$f$的相对极大值点。

对于单个变量的函数，我们有二阶导数检测法（定理 A.5），可以用来处理较复杂的函数。对于两个变量的函数，我们也有一个类似的定理。

【定理 A.7】二阶偏导数检测：假设有函数f，它具有临界点(x_0,y_0)，并且在包含该点的一些圆内，函数具有连续的二阶偏导数。如果我们设

$$D=\frac{\partial^2 f}{\partial x^2}(x_0,y_0)\frac{\partial^2 f}{\partial y^2}(x_0,y_0)-(\frac{\partial^2 f}{\partial y\partial x}(x_0,y_0))^2$$

那么下面的陈述成立：

i．如果$D>0$且$\frac{\partial^2 f}{\partial x^2}(x_0,y_0)>0$，那么函数$f$在$(x_0,y_0)$具有相对极小值。
ii．如果$D>0$且$\frac{\partial^2 f}{\partial x^2}(x_0,y_0)<0$，那么函数$f$在$(x_0,y_0)$具有相对极大值。
iii．如果$D<0$，那么函数f在(x_0,y_0)具有一个鞍点。
iv．如果$D=0$，那么结果不确定。■

下面是一个例子。考虑函数

$$f(x,y)=2y^2x-yx^2+4xy$$

我们首先计算它的一阶偏导数

$$\frac{\partial f}{\partial x}=4y-2xy+2y^2$$

$$\frac{\partial f}{\partial y}=4x-x^2+4xy$$

如果我们取它们的右式，并且同时计算x和y，可得

$$\{\{x\to 0,y\to -2\},\{x\to 0,y\to 0\},\{x\to \frac{4}{3},y\to -\frac{2}{3}\},\{x\to 4,y\to 0\}\}$$

其二阶偏导数为

$$\frac{\partial^2 f}{\partial x^2}(x,y)=4y-2xy+2y^2$$

$$\frac{\partial^2 f}{\partial y^2}(x,y)=4x-x^2+4xy$$

$$\frac{\partial^2 f}{\partial y\partial x}(x,y)=4-2x+4y$$

为了分析起来方便，参见表 A.10。在点（0，−2），（0，0）和（4，0）有$D<0$，因此这些都是鞍点。在点$(\frac{4}{3},-\frac{2}{3})$有$D>0$且$\frac{\partial^2 f}{\partial x}>0$，因此该点是一个极值点。该曲面如图 A.16 所示，其等高线如图 A.17 所示。

表 A.10 $f(x,y)=2y^2x-yx^2+4xy$在临界点的二阶偏微分

临界点				
(x_0,y_0)	$\frac{\partial^2 f}{\partial x^2}(x_0,y_0)$	$\frac{\partial^2 f}{\partial y^2}(x_0,y_0)$	$\frac{\partial^2 f}{\partial y\partial x}(x_0,y_0)$	$D=\frac{\partial^2 f}{\partial x^2}\frac{\partial^2 f}{\partial y^2}-(\frac{\partial^2 f}{\partial y\partial x})^2$
$(0,-2)$	4	0	−4	−16
$(0,0)$	0	0	−4	−16
$(\frac{4}{3},-\frac{2}{3})$	$\frac{4}{3}$	$\frac{16}{3}$	$-\frac{4}{3}$	$\frac{16}{3}$
$(4,0)$	0	16	−4	−16

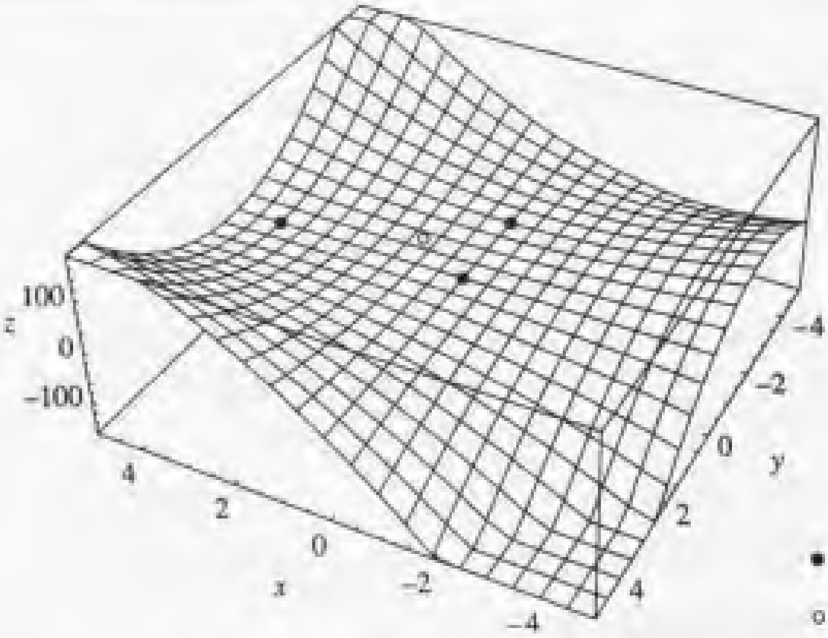


图 A.16 $2y^2x-yx^2+4xy$的图形，显示鞍点和相对极值点

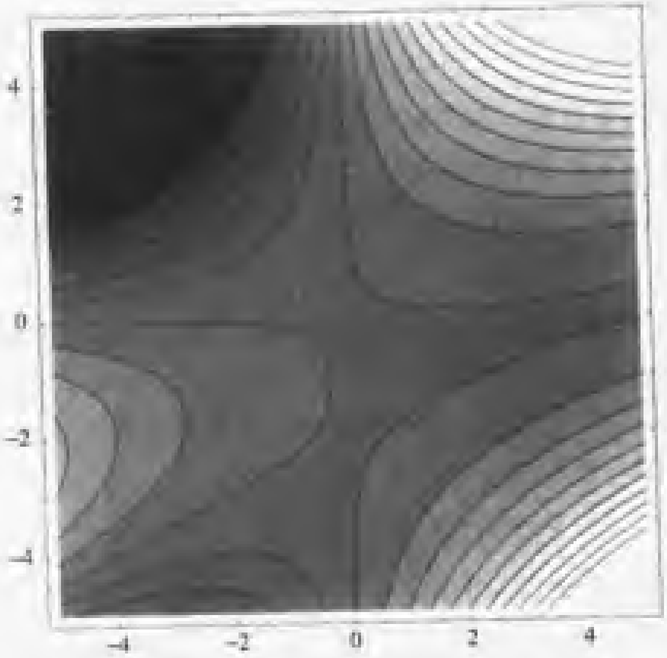


图 A.17 $2y^2x-yx^2+4xy$的等高线图

A.9.3 拉格朗日乘子

几何问题经常需要求解方程的最大值和最小值，A.9.2 节讨论了求解这类问题的方法。本节将讨论这类问题的一种特殊类型，即计算满足约束条件的最大值和最小值的问题。这类问题的规范形式描述如下：

具有一个约束条件的两个变量的极值问题：最大化或者最小化如下的函数

$$f(x, y)$$

且满足如下的约束条件

$$g(x, y) = 0$$

具有一个约束条件的三个变量的极值问题：最大化或者最小化如下的函数

$$f(x, y, z)$$

且满足如下的约束条件

$$g(x, y, z) = 0$$

满足约束条件的极值问题可以看成是一种对函数$f(x, y)$的定义域的限制。

注意，极值可能出现在区间的任一个端点，或者出现在区间内的某一点。现在我们需要两个变量的函数的模拟条件。首先介绍几个定义。

【定义 A.10】 闭集就是包含所有边界点的集合。例如，一条封闭的曲线。■

【定义 A.11】 有界集就是所有成员都被包围在一个封闭区域内的集合。■

现在我们可以进行对相关的定理的讨论了。

【定理 A.8】 极值定理——两个变量的形式：设D是一个封闭且有界的集合，并设$f(x, y)$在该集合内连续，那么存在点$(x_0, y_0), (x_1, y_1) \in D$满足$f(x, y) \leqslant f(x_0, y_0)$和$f(x, y) \geqslant f(x_1, y_1)$，$\forall (x, y) \in D$。■

可以这样来求解约束方程，即利用其他的变量来表示一个变量，并将结果代入极值方程。然后就可以应用 A.9.2 节中介绍的技术（我们还需要插入边界点，它们也可能是极值点）。然而，这样做可能行不通——约束方程可能太复杂。在这种情形中，可以运用拉格朗日乘子。

【定理 A.9】 约束极值原理——两个变量的形式：设f和g是两个在某些集合D上具有连续的一阶偏导数的两个变量的函数，D包含曲线$g(x, y) = 0$，该曲线上的任意点都满足$\nabla g \neq 0$。假设f在该约束曲线上有一个约束极值，那么该极值出现在点(x_0, y_0)，并且f和g在该点的梯度是平行的，即：

$$\nabla f(x_0, y_0) = \lambda \nabla g(x_0, y_0), \qquad \lambda \in \mathbb{R}$$

其中的λ称为拉格朗日乘子。■

一开始可能比较难以理解，因此我们在这里给出一种较通俗的解释。从 A.9.1 节中，我们知道f的梯度垂直于f的切向量：

$$\nabla f(x_0, y_0) \cdot (x'(t_0)\mathbf{i} + y'(t_0)\mathbf{j}) = 0$$

如果我们假设(x_0, y_0)确实是极值点，那么它位于约束曲线$g(x, y) = 0$上。

然而，$\nabla g(x_0, y_0)$当然也必须在点(x_0, y_0)上垂直于$g(x, y)$，因为它是函数 g 的阶层曲线。因此，$\nabla f(x_0, y_0)$和$\nabla g(x_0, y_0)$都在点(x_0, y_0)上垂直于约束曲线，因此，它们的切向量是平行的。更直接地说，两条曲线在极值点相切。

可用一个简单的例子来说明这一点。假设我们有一个方程如下的椭圆：

$$\mathcal{E}: 15x^2 + 7y^2 + 11xy = 30$$

我们要找到椭圆上最接近于原点（0，0）的点。即我们要最小化距离函数$\sqrt{x^2 + y^2}$，并满足约束条件$(x, y) \in \mathcal{E}$。集合$\mathcal{E}$是封闭且有界的，因此，由定理 A.8 可知存在一些点$P \in \mathcal{E}$满足 $f(P) \leqslant f(Q), \forall Q \in \mathcal{E}$。

我们要最小化如下的函数（我们使用平方距离以避免不必要的平方根计算）：

$$f(x, y) = x^2 + y^2$$

并满足约束条件

$$g(x, y) = 17x^2 + 8y^2 + 12xy = 100$$

根据定理 A.9，我们必定有

$$\nabla f = \lambda \nabla g$$

这样就得到一对方程

$$2x = \lambda(34x + 12y)$$

$$2y = \lambda(12x + 16y)$$

我们需要考虑如下的两种情形。

【情形 1】假设$34x + 12y \neq 0$和$12x + 16y \neq 0$。我们需要求解方程中的 λ 并使它们相等：

$$\frac{2x}{34x + 12y} = \frac{2y}{12x + 16y}$$

$$2x(12x + 16y) = 2y(34x + 12y)$$

$$12x^2 + 16xy = 34xy + 12y^2$$

$$2x^2 - 3xy - 2y^2 = 0$$

这样就得到系统

$$17x^2 + 12xy + 8y^2 = 100$$

$$2x^2 - 3xy - 2y^2 = 0$$

它们的解为（2，1），（2，−4），（−2，−1），（−2，4）。然后我们将它们代入方程：

$$f(2, 1) = 5$$

$$f(-2, -1) = 5$$

$$f(2, -4) = 20$$

$$f(-2, 4) = 20 \ \blacksquare$$

【情形 2】 $34x+12y=0$或$12x+16y=0$之一，但是在任何一种情形中，$x=y=0$都不满足椭圆方程。

通过观察这些函数的图形，几何直观有助于我们的进一步理解。图 A.18 显示了椭圆$17x^2+8y^2+12xy=100$。目标函数x^2+y^2显示在图 A.19 中，它的各种阶层曲线显示在图 A.20 中。

如果我们将椭圆和阶层曲线的图形画在一起，那么我们可以看出它们相切于点（2，1）和（−2，−1）（最小化的解），以及点（2，−4）和（−2，4）（最大化的解），如图 A.21 所示。■

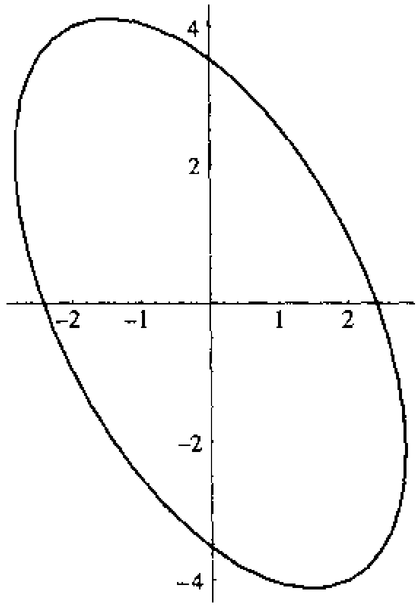


图 A.18　椭圆$17x^2+8y^2+12xy=100$的图形

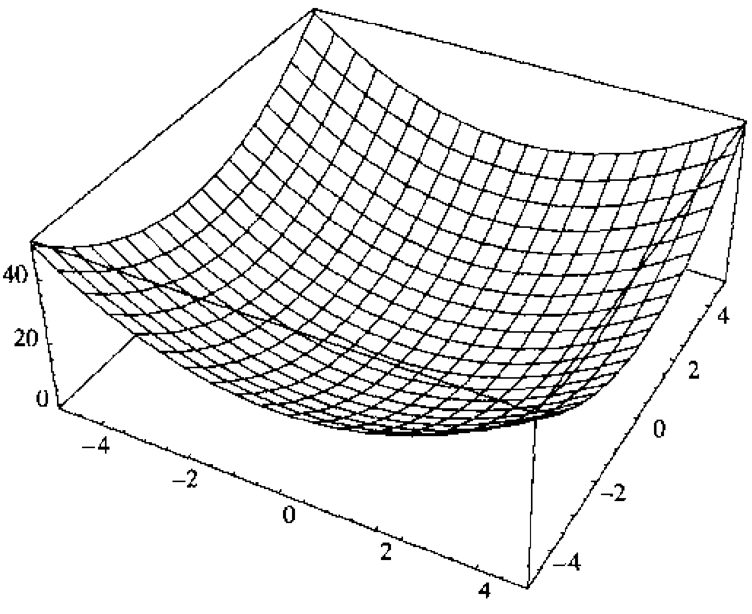


图 A.19　函数x^2+y^2的图形

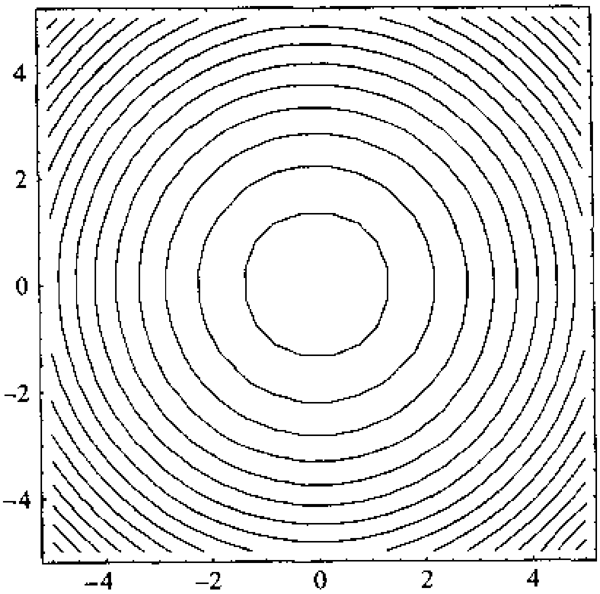


图 A.20 x^2+y^2 的阶层曲线

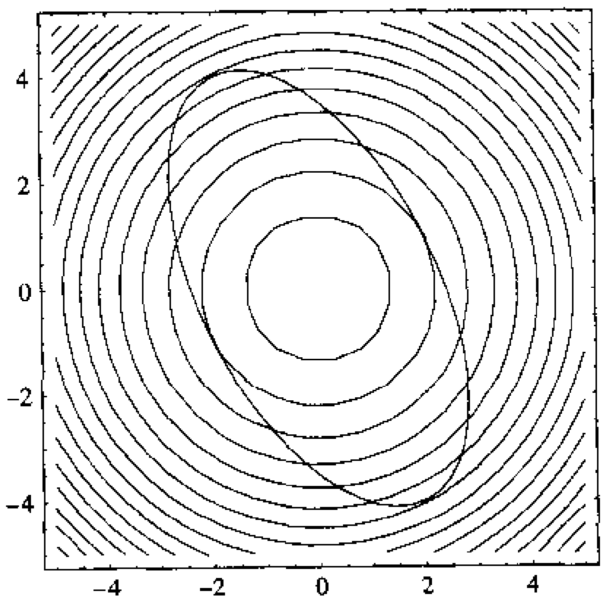


图 A.21 约束曲线和椭圆相切于最小值点

三个极值点问题

拉格朗日乘子也可用于求解三个变量的函数的满足一个或者两个约束条件的极值问题。

1. 一个约束条件

在一个约束条件的情形中，我们希望在满足约束条件 $g(x,y,z)=0$ 的最大化或者最小化函数 $f(x,y,z)$。形式为 $g(x,y,z)=0$ 的三个变量的函数的图形是三维空间中的一个曲面 S。这里的几何直观是，我们要寻找当 (x,y,z) 沿曲面 S 变化时，$f(x,y,z)$ 的最大值或者最小值。如果某一点 (x_0,y_0,z_0) 是一个球的球心，并对于所有位于该球内的 S 上的点都满足下式

$$f(x_0,y_0,z_0)\geqslant f(x,y,z)$$

则函数 $f(x,y,z)$ 在该点具有一个受约束的相对最大值。如果函数 f 的极大值是 $f(x_0,y_0,z_0)=c$，那么其阶层曲面 $f(x,y,z)=c$ 与阶层曲面 g 相切，因此，他们在该点的梯度向量在该点平行：

$$\nabla f(x_0,y_0,z_0)=\lambda\nabla g(x_0,y_0,z_0)$$

为了求解这类问题，我们必须按如下的步骤来进行处理：

（1）寻找所有满足下面的条件的值 x,y,z：

$$\nabla f(x_0,y_0,z_0)=\lambda\nabla g(x_0,y_0,z_0)$$

和

$$g(x_0,y_0,z_0)=k$$

（2）计算函数 f 在第一步中求得的点上的函数值，以确定哪些是极大值、哪些是极小值。

下面的例子将有助于说明这一点。

寻找位于球面 $x^2+y^2+z^2=36$ 上且最接近于（1，2，2）的点。即我们要寻找使距离（平方距离）函数 $f(x,y,z)=(x-1)^2+(y-2)^2+(z-2)^2$ 最小化并满足约束条件——该点必须位于球面 $g(x,y,z)=x^2+y^2+z^2=36$ 上的点 (x_0,y_0,z_0)。建立等式 $\nabla f(x,y,z)=\lambda\nabla g(x,y,z)$，可得

$$2(x-1)\mathbf{i}+2(y-2)\mathbf{j}+2(z-2)\mathbf{k}=\lambda(2x\mathbf{i}+2y\mathbf{j}+2z\mathbf{k})$$

这样得到如下的系统

$$\begin{aligned}2(x-1)&=2x\lambda\\2(y-2)&=2y\lambda\\2(z-2)&=2z\lambda\end{aligned}\tag{A.12}$$

由于这个球面的球心位于原点，并且该点的所有分量都不为零，最接近的点的任何分量都不可能具有为零的分量，因此，不需要考虑上一个例子中那样的特殊情形。我们可以将系统（A.12）改写为

$$\begin{aligned}\frac{x-1}{x}&=\lambda\\\frac{y-2}{y}&=\lambda\\\frac{z-2}{z}&=\lambda\end{aligned}\tag{A.13}$$

由前两个方程可得

$$\begin{aligned}\frac{x-1}{x}&=\frac{y-2}{y}\\xy-y&=xy-2x\\y&=2x\end{aligned}\tag{A.14}$$

而第一个和第三个方程可得

$$\frac{x-1}{x}=\frac{z-2}{z}$$
$$xz-z=xz-2x \qquad (A.15)$$
$$z=2x$$

将这些结果代入约束方程（即上述的球面方程），可得

$$9x^2=36$$

或者

$$x=\pm 2$$

如果我们将它们代入方程（A.14）和（A.15），就可得得到两个点（2，4，4）和（−2，−4，−4）。将这些值代回 f，可得 $f(2,4,4)=9$ 和 $f(-2,-4,-4)=81$，因此，我们可以得出结论：（2，4，4）是最接近的点，如图 A.22 所示。

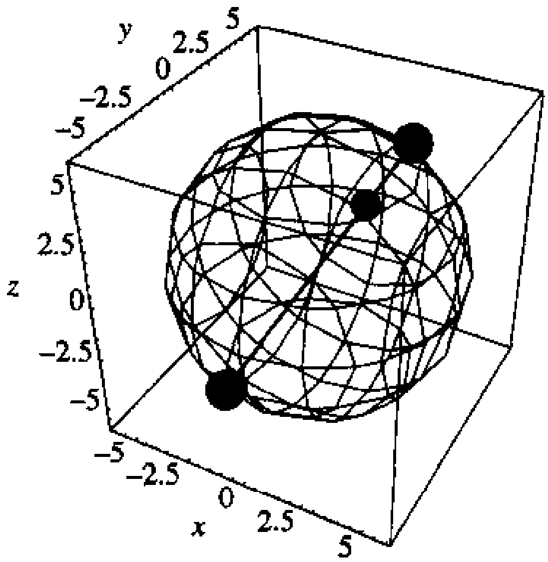


图 A.22 球面 $x^2+y^2+z^2=36$ 上距点（1，2，2）最近和最远的点

求解上述问题的另一种方法是求解一个由四个变量的方程所组成的系统

$$\frac{\partial f}{\partial x}(x_0,y_0,z_0)=\lambda\frac{\partial g}{\partial x}(x_0,y_0,z_0)$$
$$\frac{\partial f}{\partial y}(x_0,y_0,z_0)=\lambda\frac{\partial g}{\partial y}(x_0,y_0,z_0)$$
$$\frac{\partial f}{\partial z}(x_0,y_0,z_0)=\lambda\frac{\partial g}{\partial z}(x_0,y_0,z_0)$$
$$g(x_0,y_0,z_0)=0$$

或者

$$2\ (-1+x)\ =2\,x\,\lambda$$
$$2\ (-2+y)\ =2\,y\,\lambda$$
$$2\ (-2+z)\ =2\,z\,\lambda$$

$$x^2+y^2+z^2-36=0$$

可用标准技术来求解。这样我们就可以直接求得结果（与前一种方法所得的结果相同），包括拉格朗日乘子的确切值：$\{\{\lambda\to\frac{1}{2}, x\to 2, y\to 4, z\to 4\}, \{\lambda\to\frac{3}{2}, x\to -2, y\to -4, z\to -4\}\}$。

2. 两个约束条件

如果我们有一个含有三个变量的函数，就能设定两个约束条件。

【定理 A.10】 约束极值原理——具有两个约束条件的三个变量的形式：如果函数 $f(x,y,z)$ 在由曲面 $g_1(x,y,z)=0$ 和 $g_2(x,y,z)=0$ 相交所确定的约束曲线上有一个极值，那么它将出现在一个满足如下所有条件的点 (x_0,y_0,z_0)：

$$\nabla f(x_0,y_0,z_0)=\lambda_1\nabla g_1(x_0,y_0,z_0)$$

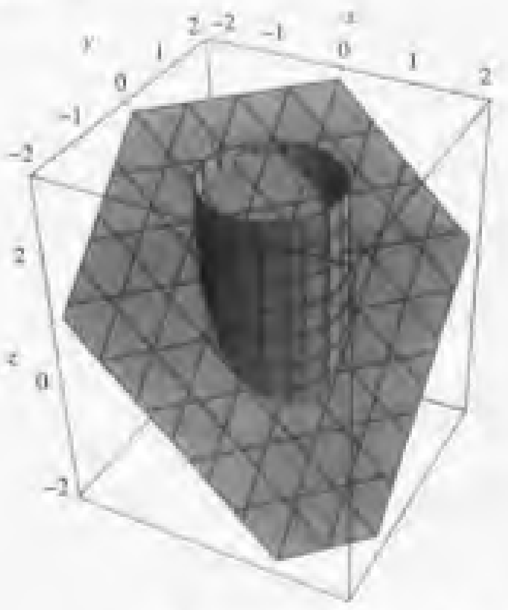
$$\nabla f(x_0,y_0,z_0)=\lambda_2\nabla g_2(x_0,y_0,z_0)$$

$$g_1(x_0,y_0,z_0)=0$$

$$g_2(x_0,y_0,z_0)=0$$

即，f 的梯度必定平行于 g_1 和 g_2 的梯度。■

我们同样可以用一个例子来说明这一定理：寻找满足约束条件 $g_1(x,y,z)=x-y+z=1$ 和 $g_2(x,y,z)=x^2+y^2=1$ 的 $f(x,y,z)=x+2y+3z=0$ 的极值。从几何意义来说，上述两个约束条件分别表示一个平面和一个圆柱面，它们的交集位于一个椭圆上。函数 f 是另一个平面，因此我们要寻找该平面在这个椭圆上的极值。图 A.23 显示了约束函数 g_1 和 g_2，它们的交集是一个椭圆。



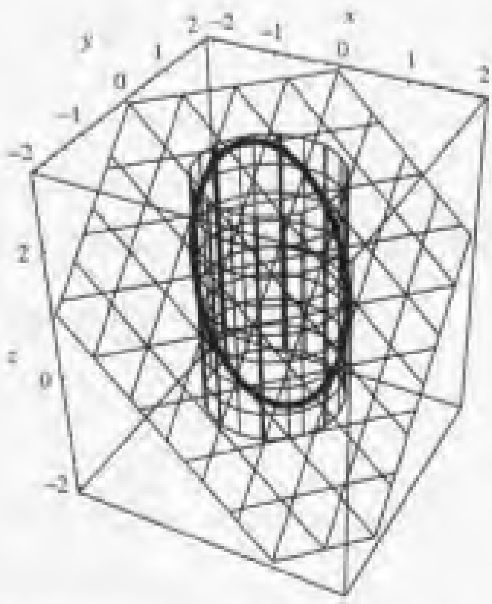


图 A.23　约束方程 $g_1(x,y,z)=x-y+z=1$ 和 $g_2(x,y,z)=x^2+y^2=1$

定理 A.10 说明，可以通过求解方程系统来寻找极值。下面是一个例子，我们的方程可以表示为

$$\frac{\partial f}{\partial x}=\lambda_1\frac{\partial g_1}{\partial x}+\lambda_2\frac{\partial g_2}{\partial x}$$

$$\frac{\partial f}{\partial y}=\lambda_1\frac{\partial g_1}{\partial y}+\lambda_2\frac{\partial g_2}{\partial y}$$

$$\frac{\partial f}{\partial z}=\lambda_1\frac{\partial g_1}{\partial z}+\lambda_2\frac{\partial g_2}{\partial z}$$

$$g_1=x-y+z-1$$

$$g_2=x^2+y^2-1$$

即为

$$1=2\,x\,\lambda_1+2\,x\,\lambda_2$$

$$2=2\,y\,\lambda_1+2\,y\,\lambda_2$$

$$3=2\,z\,\lambda_1$$

$$x^2+y^2+z^2-36=0$$

$$x^2+y^2-1=0$$

利用标准求解方法可得

$$\left\{\lambda_1\to 3,\quad \lambda_2\to\frac{-\sqrt{29}}{2},\quad x\to\frac{2}{\sqrt{29}},\quad y\to\frac{-5}{\sqrt{29}},\quad z\to\frac{29-7\sqrt{29}}{29}\right\},$$

$$\left\{\lambda_1\to 3,\quad \lambda_2\to\frac{\sqrt{29}}{2},\quad x\to\frac{-2}{\sqrt{29}},\quad y\to\frac{5}{\sqrt{29}},\quad z\to\frac{29+7\sqrt{29}}{29}\right\}$$

这些点——极大值点和极小值点——都显示在由 $g_1=0$ 和 $g_2=0$ 隐含定义的曲面相交所确定的约束曲线上，如图 A.24 所示。

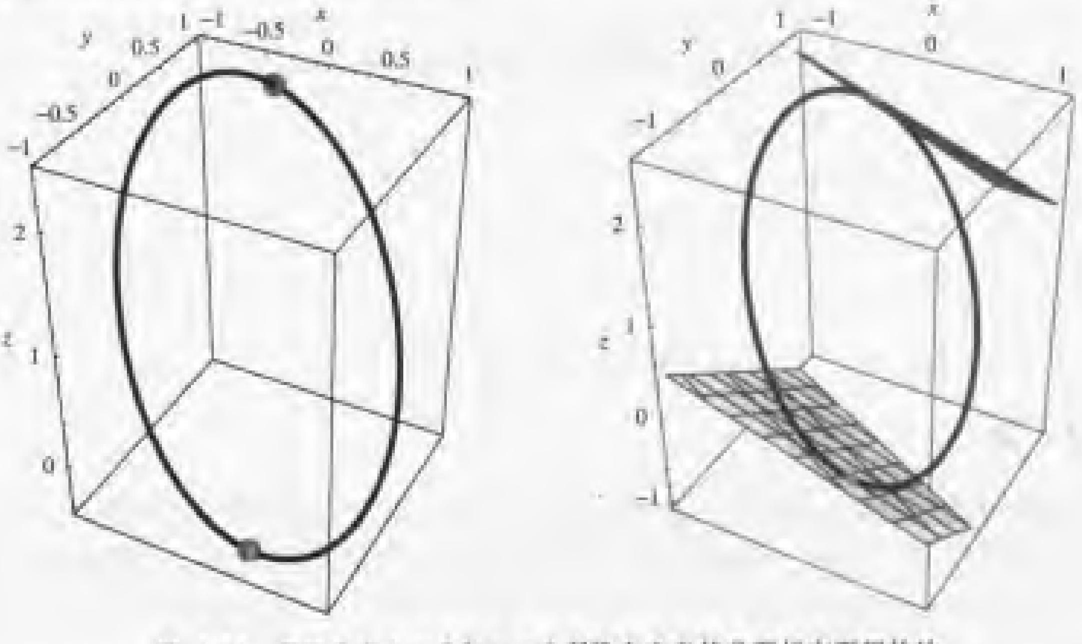


图 A.24 显示为由 $g_1=0$ 和 $g_2=0$ 所隐含定义的曲面相交所得的约束曲线上点的 f 的极值点，以及 f 在这些极值点的水平集

附录B　三 角 几 何

B.1　导言

本附录复习了一些非常有用的基本三角几何知识，同时提供了常用的定义和三角函数关系的很方便的参考。

B.1.1　术语

一般地，三角几何研究的是平面上的直线（或者更合适地说是半线或射线）之间的角。一般约定，开始测量角的射线称为起始边，而终止测量角θ的射线称为终止边。如果按逆时针方向测量，则角为正，而按顺时针方向测量，则角为负。

与角相关的两条射线的端点叫做顶点。顶点位于原点且起始边位于正x轴的角是标准位置的角（参见图 B.1）。

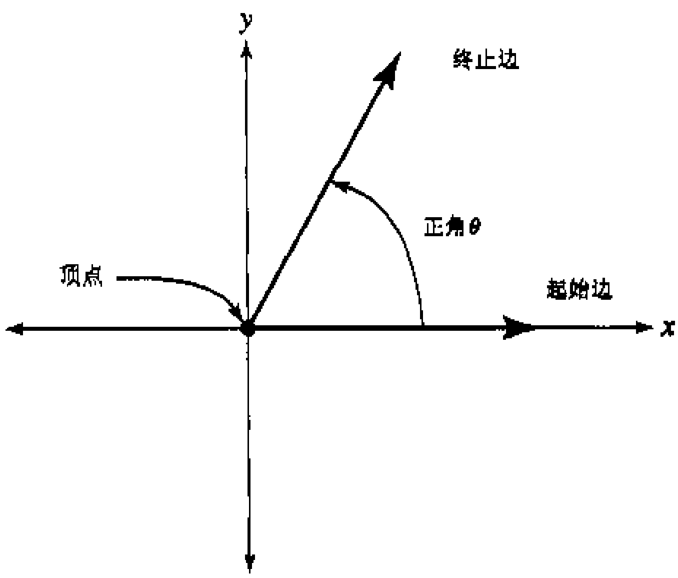


图 B.1　角的标准术语

B.1.2　角

两条射线之间的角θ可用角度或者弧度来表示。在非正式或者大众化应用中，度更常用一些，而在技术应用中，弧度更常用。如果我们设想一条射线从起始边开始扫描，继续扫描直到终止边与起始边重合，那么射线的终点轨迹形成了一个完整的圆。对应于这个圆的角被定义为 360 度或者 2π 弧度：

$$
\begin{aligned}
1^\circ &= \frac{\pi}{180}\ \text{rad} \\
&\approx 0.017453\ \text{rad}
\end{aligned}
$$

并且

$$
\begin{aligned}
1\ \text{rad} &= \frac{180^\circ}{\pi} \\
&\approx 57.2958^\circ \\
&\approx 57^\circ 17' 44.8''
\end{aligned}
$$

一般地，如果没有给出单位，角的测量都是按弧度计算的。

弧度的定义是很精确的——这就是为什么一个整圆等价于2π弧度的原因。首先，我们必须定义弧长：如果我们追踪一个点在圆上从 A 移动到 B 的路径，那么该点所移动的距离是圆的一段弧，其长度就是弧长，一般表示为 s（如图 B.2 所示）。

定义

$$\theta = \frac{s}{r}$$

为角θ的弧度（如图 B.3 所示），并考虑单位圆（其中 r=1）。我们知道，π 的定义是一个圆的周长与其直径（即 $2r$）之比。结果是，一个整圆的弧度必定是2π。

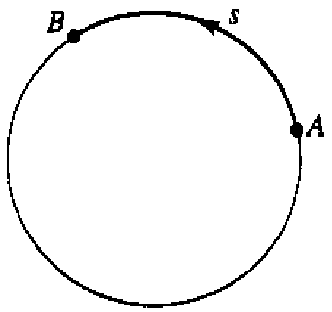


图 B.2 弧长的定义

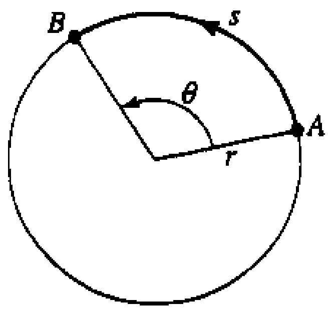


图 B.3 弧度的定义

B.1.3 变换示例

问题：将 129°变换为弧度。

解：根据定义，

$$1^\circ = \frac{\pi}{180}\ \text{rad}$$

进行一些简单的算术运算：

$$
\begin{aligned}
129^\circ &= \frac{\pi}{180}\cdot 129\ \text{rad} \\
&= \frac{129}{180}\pi\ \text{rad} \\
&\approx 2.2514728\ \text{rad}
\end{aligned}
$$

问题：将 5 弧度变换为度。

解：根据定义，

$$1\ \text{rad} = \left(\frac{180}{\pi}\right)^{\circ}$$

进行一些简单的算术运算：

$$\begin{aligned} 5\ \text{rad} &= \left(5 \cdot \frac{180}{\pi}\right)^{\circ} \\ &= \left(\frac{900}{\pi}\right)^{\circ} \\ &\approx 286.47914^{\circ} \end{aligned}$$

B.2　三角函数

角θ（正的、锐角）的标准三角函数正弦（sine）、余弦（cosine）、正切（tangent）、余割（cosecant）、正割（secant）和余切（cotangent）可用一个直角三角形的边的长度比来定义（参见图 B.4）：

$$\sin\theta = \frac{\theta\text{的对边}}{\text{斜边}} = \frac{y}{r}$$

$$\cos\theta = \frac{\theta\text{的邻边}}{\text{斜边}} = \frac{x}{r}$$

$$\tan\theta = \frac{\theta\text{的对边}}{\theta\text{的邻边}} = \frac{y}{x}$$

$$\csc\theta = \frac{\text{斜边}}{\theta\text{的对边}} = \frac{r}{y}$$

$$\sec\theta = \frac{\text{斜边}}{\theta\text{的邻边}} = \frac{r}{x}$$

$$\cot\theta = \frac{\theta\text{的邻边}}{\theta\text{的对边}} = \frac{x}{y}$$

研究上述的定义，可得如下的结论：

$$\csc\theta = \frac{1}{\sin\theta}$$

$$\sec\theta = \frac{1}{\cos\theta}$$

$$\cot\theta = \frac{1}{\tan\theta}$$

$$\tan\theta = \frac{\sin\theta}{\cos\theta}$$

$$\cot\theta = \frac{\cos\theta}{\sin\theta}$$

记住上述公式的一种方便的记忆法是用短语“soh cah toa”来表示“正弦（*s*ine）等于对边（*o*pposite）与斜边（*h*ypotenuse）之比，余弦（*c*osine）等于邻边（*a*djacent）与斜边（*h*ypotenuse）之比，正切（*t*angent）等于对边（*o*pposite）与邻边（*a*djacent）之比”；而余割、正割和余切可以简单地记成上述三个基本函数的倒数。

然而应该注意，上述定义仅仅对位于标准位置的锐角有效。一种更完整和正式的定义集如下：对于共用一个顶点的任意射线对，定义一种坐标系统变换，使得该角位于标准位置；然后构建一个圆心位于原点的单位圆，并标记终止边与圆相交的点（如图 B.5 所示）。

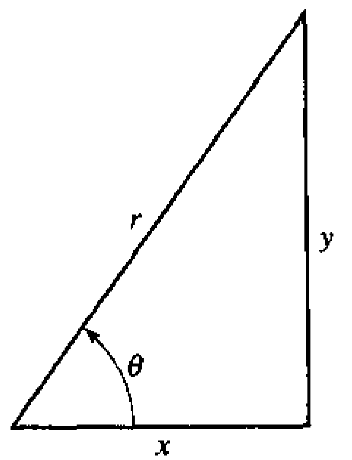


图 B.4 直角三角形的边的比值可用于定义三角函数

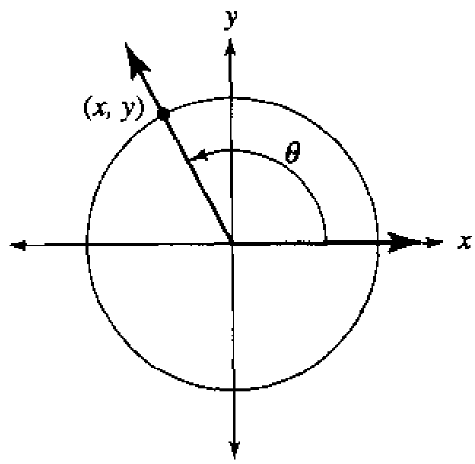


图 B.5 三角函数的一般定义

这样，我们具有如下的定义：

$$\begin{aligned}
\sin\theta &= y = \frac{y}{1} = \frac{y}{r} \\
\cos\theta &= x = \frac{x}{1} = \frac{x}{r} \\
\tan\theta &= \frac{y}{x} \\
\csc\theta &= \frac{1}{y} = \frac{r}{y} \\
\sec\theta &= \frac{1}{x} = \frac{r}{x} \\
\cot\theta &= \frac{x}{y}
\end{aligned}$$

注意，对于锐角来说，等于 1 的半径对应于长度为 1 的斜边，因此，上式的最后一列可被认为等价于前面的定义。还应注意，对于导致 x 或 y 为零的角，被 x 或 y 除的三角几何函数将分别是无定义的（如表 B.1 所示）。

表 B.1　一些常用角度的三角函数值

	$\sin\theta$	$\cos\theta$	$\tan\theta$	$\csc\theta$	$\sec\theta$	$\cot\theta$
$0 = 0^\circ$	0	1	0	—	1	—
$\pi/12 = 15^\circ$	$\frac{1}{4}(\sqrt{6}-\sqrt{2})$	$\frac{1}{4}(\sqrt{6}+\sqrt{2})$	$2-\sqrt{3}$	$\frac{4}{\sqrt{6}-\sqrt{2}}$	$\frac{4}{\sqrt{6}+\sqrt{2}}$	$2+\sqrt{3}$
$\pi/6 = 30^\circ$	$1/2$	$\sqrt{3}/2$	$1/\sqrt{3}$	2	$2/\sqrt{3}$	$\sqrt{3}$
$\pi/4 = 45^\circ$	$1/\sqrt{2}$	$1/\sqrt{2}$	1	$\sqrt{2}$	$\sqrt{2}$	1
$\pi/3 = 60^\circ$	$\sqrt{3}/2$	$1/2$	$\sqrt{3}$	$2/\sqrt{3}$	2	$1/\sqrt{3}$
$5\pi/12 = 75^\circ$	$\frac{1}{4}(\sqrt{6}+\sqrt{2})$	$\frac{1}{4}(\sqrt{6}-\sqrt{2})$	$2+\sqrt{3}$	$\frac{4}{\sqrt{6}+\sqrt{2}}$	$\frac{4}{\sqrt{6}-\sqrt{2}}$	$2-\sqrt{3}$
$\pi/2 = 90^\circ$	1	0	—	1	—	0
$7\pi/12 = 105^\circ$	$\frac{1}{4}(\sqrt{6}+\sqrt{2})$	$-\frac{1}{4}(\sqrt{6}-\sqrt{2})$	$-2-\sqrt{3}$	$\frac{4}{\sqrt{6}+\sqrt{2}}$	$-\frac{4}{\sqrt{6}-\sqrt{2}}$	$-2+\sqrt{3}$
$2\pi/3 = 120^\circ$	$\sqrt{3}/2$	$-1/2$	$-\sqrt{3}$	$2/\sqrt{3}$	-2	$-1/\sqrt{3}$
$3\pi/4 = 135^\circ$	$1/\sqrt{2}$	$-1/\sqrt{2}$	-1	$\sqrt{2}$	$-\sqrt{2}$	-1
$5\pi/6 = 150^\circ$	$1/2$	$-\sqrt{3}/2$	$-1/\sqrt{3}$	2	$-2/\sqrt{3}$	$-\sqrt{3}$
$11\pi/12 = 165^\circ$	$\frac{1}{4}(\sqrt{6}-\sqrt{2})$	$-\frac{1}{4}(\sqrt{6}+\sqrt{2})$	$-2-\sqrt{3}$	$\frac{4}{\sqrt{6}-\sqrt{2}}$	$-\frac{4}{\sqrt{6}+\sqrt{2}}$	$-2-\sqrt{3}$
$\pi = 180^\circ$	0	-1	0	—	-1	—
$3\pi/2 = 270^\circ$	-1	0	—	-1	—	0
$2\pi = 360^\circ$	0	1	0	—	1	—

从上面的等式中可以推导出非常有用的结论：如果我们有一个位于标准位置的角，那么其终止边与单位圆相交于点$(x, y) = (\cos\theta, \sin\theta)$。而且，所有的基础三角函数都有一定的几何解释，如图 B.6 所示。

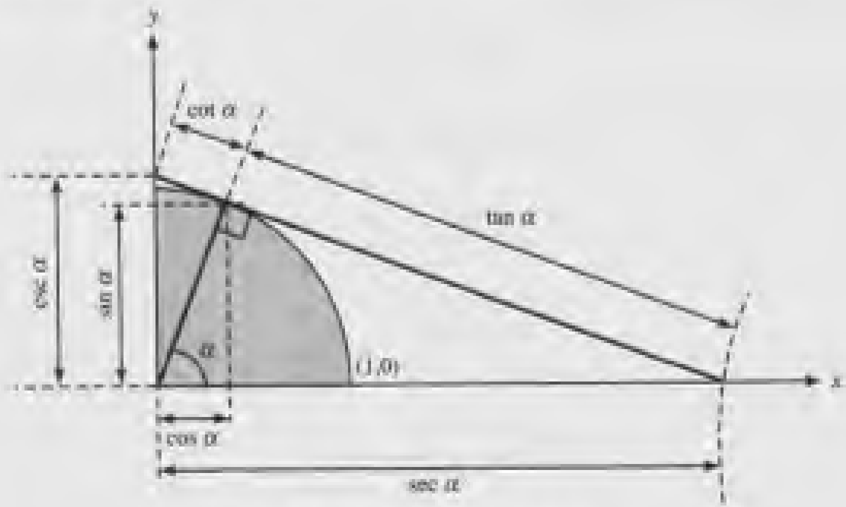


图 B.6　三角函数的几何解释

B.2.1　指数形式定义

令人感兴趣的是，基本的三角函数可用e来严格定义：

$$\sin\alpha = \frac{e^{i\alpha} - e^{-i\alpha}}{2i}$$

$$\cos\alpha = \frac{e^{i\alpha} + e^{-i\alpha}}{2}$$

$$\tan\alpha = -i\frac{e^{i\alpha} - e^{-i\alpha}}{e^{i\alpha} + e^{-i\alpha}}$$

$$= -i\frac{e^{2i\alpha} - 1}{e^{2i\alpha} + 1}$$

其中$i = \sqrt{-1}$。

e 值本身也可用三角函数来定义：

$$e^{i\alpha} = \cos\alpha + i\sin\alpha$$

B.2.2 定义域和值域

表 B.2 显示了基础三角函数的值域和定义域。从图 B.7 中可以看出，定义域一般是无穷的。除了正弦和余弦函数，其余函数的定义域都不包括一个特殊的离散的值。

表 B.2 三角函数的定义域和值域

	定义域	值域
sin	$-\infty < x < \infty$	$-1 \leqslant y \leqslant 1$
cos	$-\infty < x < \infty$	$-1 \leqslant y \leqslant 1$
tan	$-\infty < x < \infty, x \neq \frac{\pi}{2} + n\pi$	$-\infty < y < \infty$
sec	$-\infty < x < \infty, x \neq \frac{\pi}{2} + n\pi$	$\lvert y \rvert \geqslant 1$
csc	$-\infty < x < \infty, x \neq n\pi$	$\lvert y \rvert \geqslant 1$
cot	$-\infty < x < \infty, x \neq n\pi$	$-\infty < y < \infty$

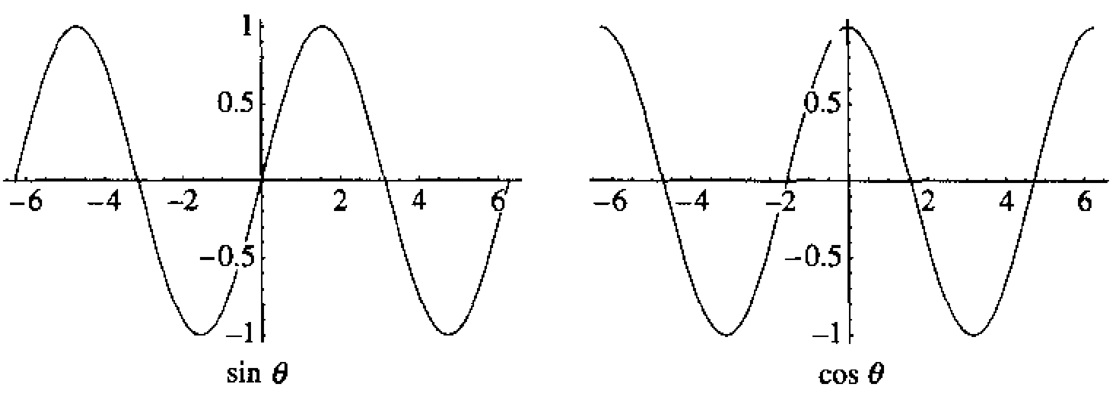


图 B.7 基础三角函数的图形

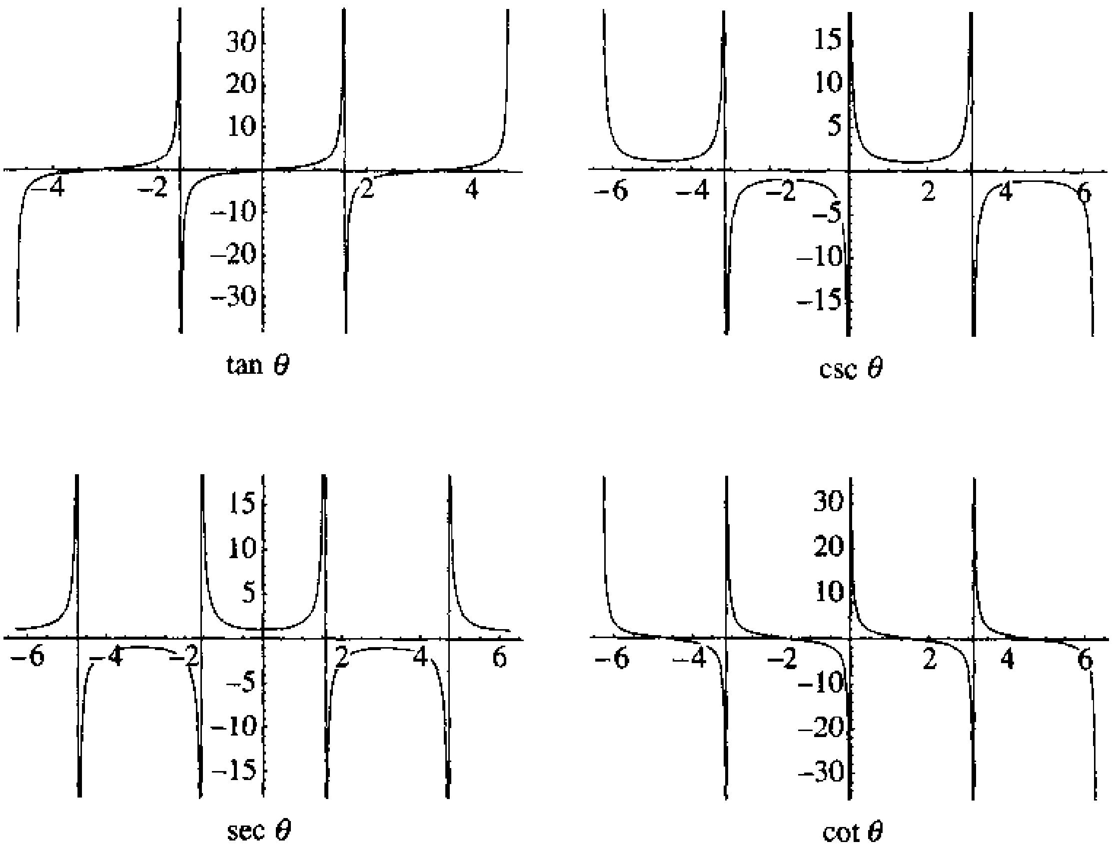


图 B.7　基础三角函数的图形（续）

B.2.3　三角函数的图形

图 B.7 显示了每一种基础三角函数的图形的片断。

B.2.4　三角函数的导数

函数 f 的导数表示为 f'，可以定义为

$$f'(x) = \lim_{n \to 0} \frac{f(x+h) - f(x)}{h}$$

将每一种三角函数代入导数的定义式，并利用三角几何的加法公式进行一些简单的运算，以进行简化，就能得到三角函数的导数。例如，

$$\begin{aligned}\frac{d}{dx}(\sin x) &= \lim_{n \to 0} \frac{\sin(x+h) - \sin x}{h} \\ &= \lim_{n \to 0} \frac{\sin x \cos h + \cos x \sin h - \sin x}{h} \\ &= \lim_{n \to 0} \left[\sin x \left(\frac{\cos h - 1}{h}\right) + \cos x \left(\frac{\sin h}{h}\right)\right]\end{aligned}$$

$\sin x$ 和 $\cos x$ 项相对于 h 是常数，因此

$$\lim_{n\to 0} \sin x = \sin x$$

$$\lim_{n\to 0} \cos x = \cos x$$

因而有

$$\frac{d}{dx}(\sin x) = \sin x \cdot \lim_{n\to 0}\left(\frac{\cos h - 1}{h}\right) + \cos x \cdot \lim_{n\to 0}\left(\frac{\sin h}{h}\right)$$

可以证明

$$\lim_{n\to 0}\frac{\cos h - 1}{h} = 0$$

$$\lim_{n\to 0}\frac{\sin h}{h} = 1$$

所以有

$$\frac{d}{dx}[\sin x] = \cos x$$

可用类似的方法来计算 cos x 的导数，可得

$$\frac{d}{dx}[\cos x] = -\sin x$$

其余的三角函数可以定义为正弦和余弦函数的简单组合（参见 B.2.1 节）。我们可以简单地使用微积分的商原理（quotient rule）并求得

$$\frac{d}{dx}[\tan x] = \sec^2 x$$

$$\frac{d}{dx}[\cot x] = -\csc^2 x$$

$$\frac{d}{dx}[\sec x] = \sec x \tan x$$

$$\frac{d}{dx}[\csc x] = -\csc x \cot x$$

B.2.5 积分

$$\int \sin u \, du = -\cos u + C$$

$$\int \cos u \, du = \sin u + C$$

$$\int \tan u \, du = \ln|\sec u| + C$$

$$\int \cot u \, du = -\ln|\sin u| + C$$

$$\int \sec u\, du = \ln|\sec u + \tan u| + C$$

$$= \ln\left|\tan\left(\frac{1}{4}\pi + \frac{1}{2}u\right)\right| + C$$

$$\int \csc u\, du = \ln|\csc u - \cot u| + C$$

$$= \ln\left|\tan\frac{1}{2}u\right| + C$$

B.3　三角恒等式和定律

考虑图 B.4，并应用毕达哥拉斯定理（译者注：即勾股定理），可得如下的关系：

$$x^2 + y^2 = r^2$$

如果进行一些算术运算，并根据正弦和余弦函数的定义式，可得

$$x^2 + y^2 = r^2$$

$$\frac{x^2 + y^2}{r^2} = 1 \qquad \text{两边除以 } r^2$$

$$\sin^2\theta + \cos^2\theta = 1 \qquad \text{使用正弦和余弦的定义式}$$

下面的一组恒等式涉及角的反转。考虑一个角 θ 及 B.2.1 节中的定义。如果我们考虑一个角 $-\theta$，我们可以看到，该角的终止边与单位圆相交的点的 x 坐标与角 θ 的终止边与单位圆的交点的 x 坐标相同，但是它们的 y 坐标是相反的（观察图 B.5）。于是，利用 B.2.1 节中的定义，我们可以定义

$$\sin(-\theta) = \frac{-y}{r}$$

$$= -\frac{y}{r}$$

$$= -\sin\theta$$

以及

$$\cos(-\theta) = \frac{x}{r}$$

$$= \cos\theta$$

和

$$\tan(-\theta) = \frac{-y}{r}$$

$$= -\frac{y}{r}$$

$$= -\tan(\theta)$$

B.3.1 周期

图 B.7 中的三角函数图形表明三角函数都是周期性的，而 B.2 节中给出的定义也揭示出所有共用终止边的角的三角函数值都相同。因此，通过定义可以证明三角函数确实是周期性的，而通过观察可知一个圆由2π弧度构成，因此

$$\sin\theta = \sin(\theta \pm 2n\pi)$$

$$\cos\theta = \cos(\theta \pm 2n\pi)$$

$$\csc\theta = \csc(\theta \pm 2n\pi)$$

$$\sec\theta = \sec(\theta \pm 2n\pi)$$

对所有 $n=\ldots, -2, -1, 0, 1, 2, \ldots$ 成立。然而，正切和余切函数具有周期π：

$$\tan\theta = \tan(\theta \pm n\pi)$$

$$\cot\theta = \cot(\theta \pm n\pi)$$

对所有 $n=\ldots, -2, -1, 0, 1, 2, \ldots$ 成立。

B.3.2 定理

本节讨论关于普通三角形的边、角和函数之间的关系的三个定理。

1．正弦定理

正弦定理说明了基础三角函数之间的一种关系，以及它们与普通三角形（不仅仅是直角三角形，参见图 B.8）之间的关系。非正式地说，正弦定理说明，对于任意三角形，它的每一条边与其对角的正弦之比都相等，即

$$\frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma} = 2r$$

其中 r 是三角形的外接圆半径（通过三角形的三个顶点的圆的半径）。

下面的证明摘自 Ronald Goldman（1987）：

$$2\,\text{Area}(\triangle ABC) = \|(A-B)\times(C-B)\| = ca\sin\beta$$

$$2\,\text{Area}(\triangle BCA) = \|(B-C)\times(A-C)\| = ab\sin\gamma$$

$$2\,\text{Area}(\triangle CAB) = \|(C-A)\times(B-A)\| = bc\sin\alpha$$

$$\therefore ca\sin\beta = ab\sin\gamma = bc\sin\alpha$$

$$\therefore \frac{a}{\sin\alpha} = \frac{b}{\sin\beta} = \frac{c}{\sin\gamma}$$

与外接圆的半径之间的关系也很容易证明。再考虑我们的三角形$\triangle ABC$，选取任一个顶点，从该点画一条通过外接圆圆心（通过三角形的三个顶点的圆的圆心）的直线，并与该圆相交与点 D（如图 B.9 所示）。

我们知道$\angle ADC$是一个直角，因为它直对着一个半圆。通过正弦函数的定义，可得

$$\sin \delta = \frac{b}{AD}$$

然而，$\delta = \beta$，因为它们都直对着同一条弧$\widehat{AC}$。因此，

$$\sin \delta = \sin \beta$$

代入，可得

$$\sin \beta = \frac{b}{AD}$$

可是，由于AD通过外接圆圆心，即$AD = 2r$，因此我们有

$$\sin \beta = \frac{b}{2r}$$

移项，可得

$$2r = \frac{b}{\sin \beta}$$

结合前面的证明，我们可以得出如下的结论

$$\frac{a}{\sin \alpha} = \frac{b}{\sin \beta} = \frac{c}{\sin \gamma} = 2r$$

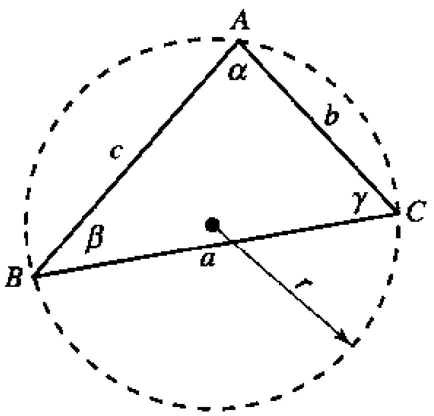


图 B.8　正弦定理

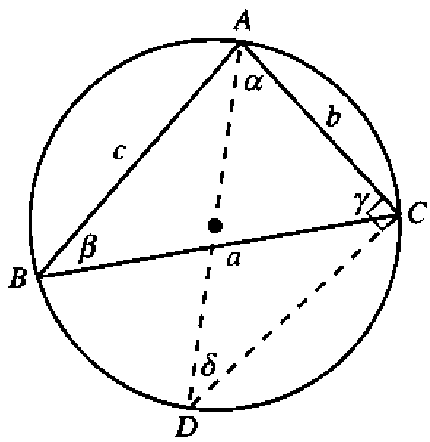


图 B.9　正弦定理的证明

2. 余弦定理

余弦定理是另一种常用的关系，可以看成是毕达哥拉斯定理（译者注：即勾股定理）扩展到所有三角形的一般形式。

对于任意边为 a，b 和 c 的三角形，如果角θ所对的边为 c，那么

$$c^2 = a^2 + b^2 - 2ab\cos\theta$$

下面的证明也摘自 Goldman（1987）：

$$\begin{aligned} c^2 &= \|B - A\|^2 \\ &= (B - A)\cdot(B - A) \\ &= [(B - C) + (C - A)]\cdot[(B - C) + (C - A)] \end{aligned}$$

$$= (B - C) \cdot (B - C) + (C - A) \cdot (C - A) - 2(A - C)(B - C)$$

$$= \|B - C\|^2 + \|C - A\|^2 - 2\|A - C\|\|B - C\| \cos C$$

$$= a^2 + b^2 - 2ab \cos C$$

3. 正切定理

正切定理说明了对于任意三角形，两条边之差与它们的和之比等于它们所对的角的差的一半和它们所对的角的和的一半的正切之比。同样参见图B.8，我们有

$$\frac{a+b}{a-b} = \frac{\tan \frac{\alpha+\beta}{2}}{\tan \frac{\alpha-\beta}{2}}$$

下面列出了摘自“数学论坛”网站（*http://forum.swarthmore.edu/dr.math*）的证明。考虑正弦函数的加法和减法公式：

$$\sin(t+u) = \sin(t)\cos(u) + \cos(t)\sin(u)$$

$$\sin(t-u) = \sin(t)\cos(u) - \cos(t)\sin(u)$$

分别相加和相减，可得下面的式子：

$$\sin(t+u) + \sin(t-u) = 2\sin(t)\cos(u)$$

$$\sin(t-u) + \sin(t-u) = 2\cos(t)\sin(u)$$

如果我们设$t = (\alpha+\beta)/2$和$u = (\alpha-\beta)/2$，那么有$t+u=\alpha$和$t-u=\beta$，这样可得

$$\sin(\alpha) + \sin(\beta) = 2\sin((\alpha+\beta)/2)\cos((\alpha-\beta)/2)$$

$$\sin(\alpha) + \sin(\beta) = 2\cos((\alpha+\beta)/2)\sin((\alpha-\beta)/2)$$

那么，我们就能求这两个方程之比，可得

$$\frac{\tan((\alpha+\beta)/2)}{\tan((\alpha-\beta)/2)} = \frac{\sin((\alpha+\beta)/2)\cos((\alpha-\beta)/2)}{\cos((\alpha+\beta)/2)\sin((\alpha-\beta)/2)}$$

$$= \frac{2\sin((\alpha+\beta)/2)\cos((\alpha-\beta)/2)}{2\cos((\alpha+\beta)/2)\sin((\alpha-\beta)/2)}$$

$$= \frac{\sin(\alpha)+\sin(\beta)}{\sin(\alpha)-\sin(\beta)}$$

根据正弦定理，上式等于

$$\frac{a+b}{a-b}$$

B.3.3 公式

本节介绍了一系列的基础公式，在求解几何问题时，有时需要用到它们。

1. 摩尔魏特公式

如果你正在求解的问题要求计算一个三角形的顶点和（或）边，已知其他（足够的）关于三角形的信息，那么就可以利用正弦、余弦和正切定理等说明的三角形的边角关系来

求解这类问题。摩尔魏特公式和牛顿公式可用来检验这类问题，因为这两个公式都与三角形的三个顶点和三条边有关。

$$\frac{b-c}{a}=\frac{\sin\frac{B-C}{2}}{\cos\frac{A}{2}}$$

2. 牛顿公式

$$\frac{b+c}{a}=\frac{\cos\frac{B-C}{2}}{\sin\frac{A}{2}}$$

3. 面积公式

对于直角三角形，很容易证明其面积公式为 $A=1/2\times$底$\times$高。然而，对于一般的三角形却没有这样方便的公式。可以使用一个更一般的公式

$$A=\frac{bc\sin\alpha}{2}=\frac{ac\sin\beta}{2}=\frac{ab\sin\gamma}{2}$$

4. 两角和与差的三角函数公式

你可能经常遇到这样的问题，即已知两个角的三角函数值，要求解这两个角的和及差的三角函数值。进行这种计算的公式如下

$$\sin(\alpha+\beta)=\sin\alpha\cos\beta+\cos\alpha\sin\beta$$

$$\sin(\alpha-\beta)=\sin\alpha\cos\beta-\cos\alpha\sin\beta$$

$$\cos(\alpha+\beta)=\cos\alpha\cos\beta-\sin\alpha\sin\beta$$

$$\cos(\alpha-\beta)=\cos\alpha\cos\beta+\sin\alpha\sin\beta$$

$$\tan(\alpha+\beta)=\frac{\tan\alpha+\tan\beta}{1-\tan\alpha\tan\beta}$$

$$\tan(\alpha-\beta)=\frac{\tan\alpha-\tan\beta}{1+\tan\alpha\tan\beta}$$

$$\cot(\alpha+\beta)=\frac{\cot\alpha\cot\beta-1}{\cot\alpha+\cot\beta}$$

$$\cot(\alpha-\beta)=\frac{\cot\alpha\cot\beta+1}{\cot\alpha-\cot\beta}$$

你也可能遇到要求两个角的三角函数的和及差的问题（即和差化积公式）：

$$\sin\alpha+\sin\beta=2\sin\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}$$

$$\sin\alpha-\sin\beta=2\cos\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}$$

$$\cos\alpha+\cos\beta=2\cos\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2}$$

$$\cos\alpha-\cos\beta=-2\sin\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2}$$

$$\tan\alpha + \tan\beta = \frac{\sin(\alpha+\beta)}{\cos\alpha\cos\beta}$$

$$\tan\alpha - \tan\beta = \frac{\sin(\alpha-\beta)}{\cos\alpha\cos\beta}$$

5. 积化和差公式

$$\sin\alpha\sin\beta = \cos\frac{\alpha-\beta}{2} - \cos\frac{\alpha+\beta}{2}$$

$$\sin\alpha\cos\beta = \sin\frac{\alpha+\beta}{2} + \sin\frac{\alpha-\beta}{2}$$

$$\cos\alpha\cos\beta = \cos\frac{\alpha-\beta}{2} + \cos\frac{\alpha+\beta}{2}$$

$$\cos\alpha\cos\beta = \cos\frac{\alpha+\beta}{2} - \cos\frac{\alpha-\beta}{2}$$

6. 二倍角公式

$$\begin{aligned}\sin 2\alpha &= 2\sin\alpha\cos\alpha \\ &= \frac{2\tan\alpha}{1+\tan^2\alpha}\end{aligned}$$

$$\begin{aligned}\cos 2\alpha &= \cos^2\alpha - \sin^2\alpha \\ &= 2\cos^2\alpha - 1 \\ &= 1 - 2\sin^2\alpha \\ &= \frac{1-\tan^2\alpha}{1+\tan^2\alpha}\end{aligned}$$

$$\tan 2\alpha = \frac{2\tan\alpha}{1-\tan^2\alpha}$$

$$\cot 2\alpha = \frac{\cot^2\alpha - 1}{2\cot\alpha}$$

7. 三倍角公式

$$\sin 3\alpha = 3\sin\alpha - 4\sin^3\alpha$$

$$\cos 3\alpha = 4\cos^3\alpha - 3\cos\alpha$$

$$\tan 3\alpha = \frac{3\tan\alpha - \tan^3\alpha}{1-3\tan^2\alpha}$$

8. 四倍角公式

$$\sin 4\alpha = 4\sin\alpha\cos\alpha - 8\sin^3\alpha\cos\alpha$$

$$\cos 4\alpha = 8\cos^4\alpha - 8\cos^2\alpha + 1$$

$$\tan 4\alpha = \frac{4\tan\alpha - 4\tan^3\alpha}{1-6\tan^2\alpha+\tan^4\alpha}$$

9. 一般倍角公式

有两种方法可以定义这种公式。第一种是利用一系列的函数的幂：

$$\sin n\alpha = n\sin\alpha\cos^{n-1}\alpha - \binom{n}{3}\sin^3\alpha\cos^{n-3}\alpha + \binom{n}{5}\sin^5\alpha\cos^{n-5}\alpha - \ldots$$

$$\cos n\alpha = \cos^n\alpha - \binom{n}{2}\sin^2\alpha\cos^{n-2}\alpha + \binom{n}{4}\sin^4\alpha\cos^{n-4}\alpha - \ldots$$

第二种则是利用较小倍数的组合：

$$\sin n\alpha = 2\sin(n-1)\alpha\cos\alpha - \sin(n-2)\alpha$$

$$\cos n\alpha = 2\cos(n-1)\alpha\cos\alpha - \cos(n-2)\alpha$$

$$\tan n\alpha = \frac{\tan(n-1)\alpha + \tan\alpha}{1-\tan(n-1)\alpha\tan\alpha}$$

10. 特殊的降幂公式

$$\sin^2\alpha = \frac{1-\cos 2\alpha}{2}$$

$$\sin^3\alpha = \frac{3\sin\alpha - \sin 3\alpha}{4}$$

$$\sin^4\alpha = \frac{3-4\cos 2\alpha + \cos 4\alpha}{8}$$

$$\cos^2\alpha = \frac{1+\cos 2\alpha}{2}$$

$$\cos^3\alpha = \frac{3\cos\alpha + \cos 3\alpha}{4}$$

$$\cos^4\alpha = \frac{3+4\cos 2\alpha + \cos 4\alpha}{8}$$

$$\tan^2\alpha = \frac{1-\cos 2\alpha}{1+\cos 2\alpha}$$

11. 一般降幂公式

$$\sin^{2n}\alpha = \binom{2n}{n}\frac{1}{2^{2n}} + \frac{1}{2^{2n-1}}\sum_{k=1}^{n}(-1)^k\binom{2n}{n-k}\cos 2k\alpha$$

$$\sin^{2n-1}\alpha = \frac{1}{2^{2n-2}}\sum_{k=1}^{n}(-1)^k\binom{2n-1}{n-k}\sin(2k-1)\alpha$$

$$\cos^{2n}\alpha = \binom{2n}{n}\frac{1}{2^{2n}} + \frac{1}{2^{2n-1}}\sum_{k=1}^{n}\binom{2n}{n-k}\cos 2k\alpha$$

$$\cos^{2n-1}\alpha = \frac{1}{2^{2n-2}}\sum_{k=1}^{n}\binom{2n-1}{n-k}\sin(2k-1)\alpha$$

12. 半角公式

$$\sin\frac{\alpha}{2}=\pm\sqrt{\frac{1-\cos\alpha}{2}}$$

$$\cos\frac{\alpha}{2}=\pm\sqrt{\frac{1+\cos\alpha}{2}}$$

$$\tan\frac{\alpha}{2}=\frac{\sin\alpha}{1+\cos\alpha}$$

$$=\frac{1-\cos\alpha}{\sin\alpha}$$

$$=\pm\sqrt{\frac{1-\cos\alpha}{1+\cos\alpha}}$$

$$\cot\frac{\alpha}{2}=\frac{\sin\alpha}{1-\cos\alpha}$$

$$=\pm\sqrt{\frac{1+\cos\alpha}{1-\cos\alpha}}$$

B.4 反三角函数

由于三角函数都是函数，因此凭直觉可以知道，它们一定有反函数。在计算机图形学中经常需要计算反三角函数的值。例如，你可能遇到一个问题，其中出现了如下的表达式

$$a=\tan b$$

当然，如果我们已知 a 要求 b，我们就需要求正切函数的反函数：

$$b=\tan^{-1}a$$

基础三角函数的反函数名称就是在基础函数名的前面加上一个“反”字：反正弦，反余弦等。有两种不同的表示方法，即

- arcsin, arccos, arctan, etc.
- $\sin^{-1}$, $\cos^{-1}$, $\tan^{-1}$, etc.

B.4.1 用 arctan 定义 arcsin 和 arccos

有意思的是，反正弦和反余弦可用仅仅与反正切有关的公式来定义：

$$\arcsin x=\arctan\frac{x}{\sqrt{1-x^2}}$$

$$\arccos x=\frac{\pi}{2}-\arctan\frac{x}{\sqrt{1-x^2}}$$

B.4.2 定义域和值域

反三角函数的定义域一般比它们的基础函数更严格。这些函数的图形说明了这一点，表 B.3 列出了精确的值。

表 B.3　反三角函数的定义域和值域

	定义域	值　域
$\sin^{-1}$	$1 \leqslant x \leqslant 1$	$-\frac{\pi}{2} \leqslant y \leqslant \frac{\pi}{2}$
$\cos^{-1}$	$-1 \leqslant x \leqslant 1$	$0 \leqslant y \leqslant \pi$
$\tan^{-1}$	$-\infty < x < \infty$	$-\frac{\pi}{2} \leqslant y \leqslant \frac{\pi}{2}$
$\sec^{-1}$	$\lvert x \rvert \geqslant 1$	$0 \leqslant y \leqslant \pi, y \neq \frac{\pi}{2}$
$\csc^{-1}$	$\lvert x \rvert \geqslant 1$	$-\frac{\pi}{2} \leqslant y \leqslant \frac{\pi}{2}, y \neq 0$
$\cot^{-1}$	$-\infty < x < \infty$	$0 < y < \pi$

B.4.3　图形

图 B.10 显示了每一种反三角函数的图形的片断。

B.4.4　导数

$$\frac{d}{dx}\left[\sin^{-1} x\right] = \frac{1}{\sqrt{1-x^2}}$$

$$\frac{d}{dx}\left[\cos^{-1} x\right] = -\frac{1}{\sqrt{1-x^2}}$$

$$\frac{d}{dx}\left[\tan^{-1} x\right] = \frac{1}{1+x^2}$$

$$\frac{d}{dx}\left[\cot^{-1} x\right] = -\frac{1}{1+x^2}$$

$$\frac{d}{dx}\left[\sec^{-1} x\right] = \frac{1}{|x|\sqrt{x^2-1}}$$

$$\frac{d}{dx}\left[\csc^{-1} x\right] = -\frac{1}{|x|\sqrt{x^2-1}}$$

B.4.5　积分

$$\int \sin^{-1} u \, du = u \sin^{-1} u + \sqrt{1-u^2} + C$$

$$\int \cos^{-1} u \, du = u \cos^{-1} u + \sqrt{1-u^2} + C$$

$$\int \tan^{-1} u \, du = u \tan^{-1} u - \ln \sqrt{1+u^2} + C$$

$$\int \cot^{-1} u \, du = u \cot^{-1} u + \ln \sqrt{1+u^2} + C$$

$$\int \sec^{-1} u \, du = u \sec^{-1} u - \ln \left| u + \sqrt{u^2-1} \right| + C$$

$$\int \csc^{-1} u\, du = u \csc^{-1} u + \ln \left| u + \sqrt{u^2 - 1} \right| + C$$

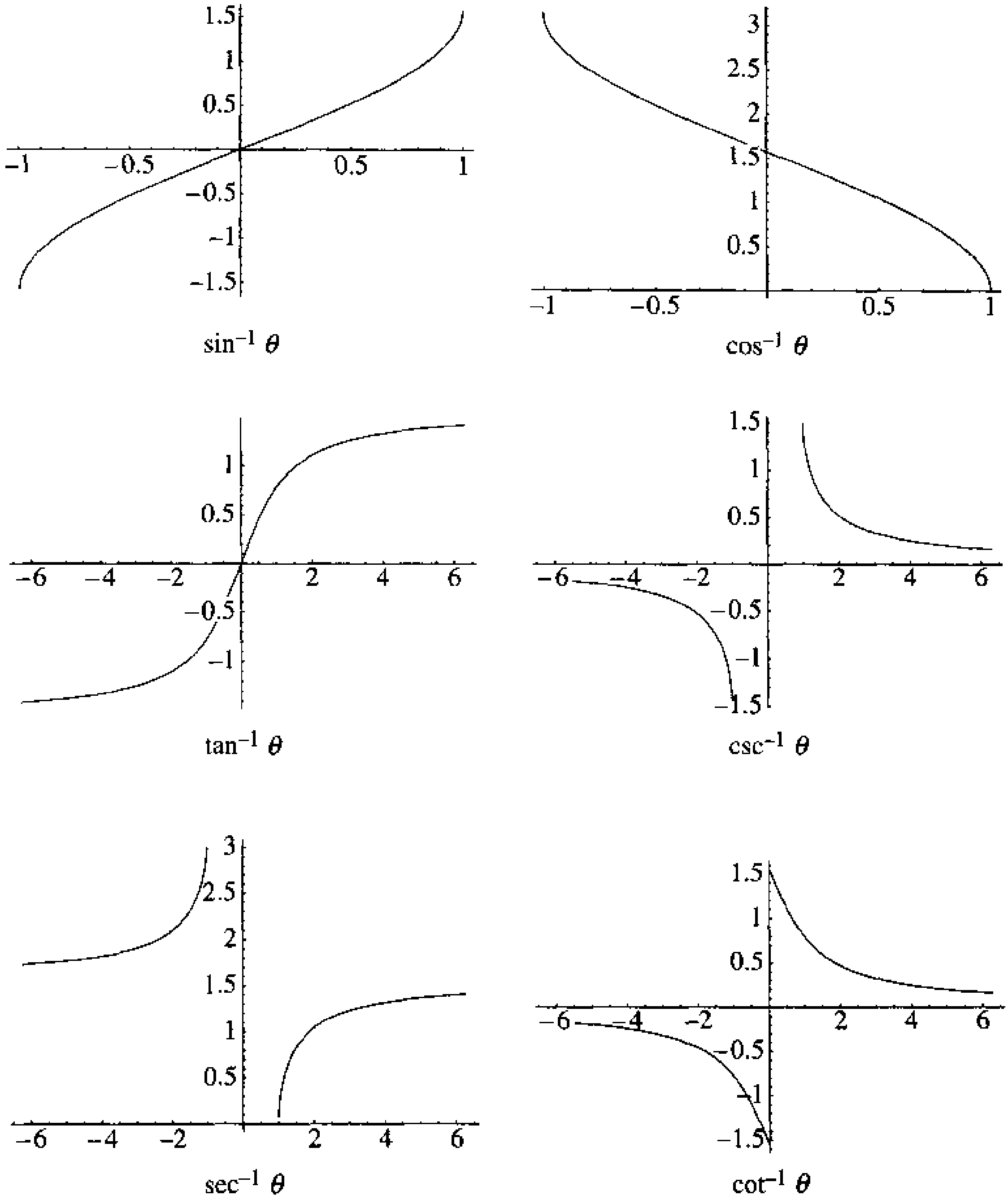


图 B.10　基础反三角函数的图形

B.5　进一步的阅读资料

Wolfram Research 公司的网站（*http://functions.wolfram.com/ElementaryFunctions*）中包含了数百页与三角函数相关的内容。含有大量三角几何知识的教材：最近在 *www.amazon.com* 上搜索包括关键字"trigonometry"（三角几何）的主题，所得结果有 682 条记录。

附录 C 几何图元基础公式

C.1 导言

本附录包含了涉及常用几何图元的不同性质的一些计算公式。

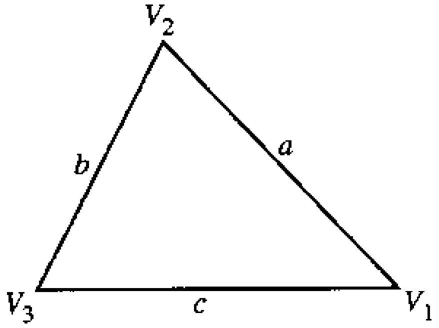
C.2 三角形

C.2.1 符号

- a，b，c：边
- α，β，γ：角
- h：高
- m：中线
- s：平分线
- $2p = a + b + c$：周长
- A：面积
- R：外接圆半径（经过三角形的三个顶点的圆的半径）
- C_R：外接圆圆心（经过三角形的三个顶点的圆的圆心）
- r：内切圆半径（与三角形的三条边相切的圆的半径）
- C_r：内切圆圆心（与三角形的三条边相切的圆的圆心）
- C_g：重心（中线的交点）
- C_{alt}：高线的交点
- V_1, V_2, V_3：顶点

C.2.2 定义

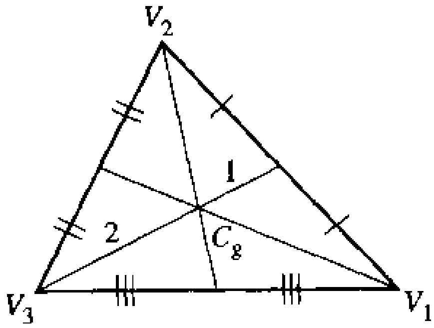
1. 周长与面积



$$2p = a + b + c$$
$$= \|V_1 - V_2\| + \|V_2 - V_3\| + \|V_3 - V_1\|$$

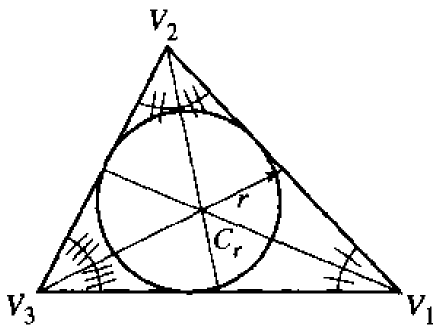
$$A = \frac{\|V_1 \times V_2 + V_2 \times V_3 + V_3 \times V_1\|}{2}$$

2. 中线的交点：重心



$$C_g = \frac{V_1 + V_2 + V_3}{3}$$

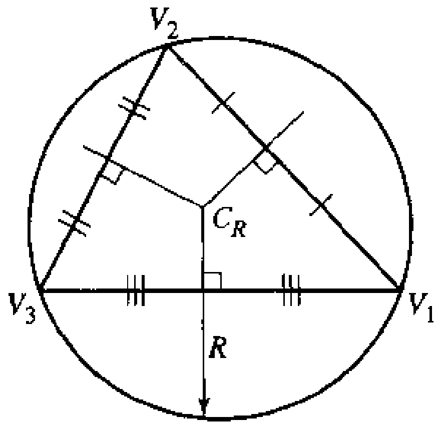
3. 角平分线的交点：内切圆的半径和圆心



$$r = \frac{2A}{2p}$$

$$C_r = \frac{\|V_2 - V_3\|V_1 + \|V_3 - V_1\|V_2 + \|V_1 - V_2\|V_3}{2p}$$

4. 边的垂直平分线的交点：外接圆的半径和圆心



$$d_{ca} = (V_3 - V_1) \cdot (V_2 - V_1)$$

$$d_{ba} = (V_3 - V_2) \cdot (V_1 - V_2)$$

$$d_{cb} = (V_1 - V_3) \cdot (V_2 - V_3)$$

$$n_1 = d_{ba}d_{cb}$$

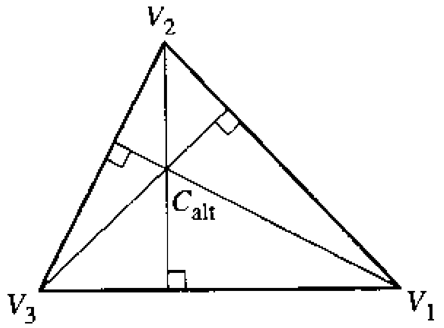
$$n_2 = d_{cb}d_{ca}$$

$$n_3 = d_{ca}d_{ba}$$

$$R = \frac{\sqrt{(d_{ca} + d_{ba})(d_{ba} + d_{cb})(d_{cb} + d_{ca}) / (n_1 + n_2 + n_3)}}{2}$$

$$C_R = \frac{(n_2 + n_3) V_1 + (n_3 + n_1) V_2 + (n_1 + n_2) V_3}{2(n_1 + n_2 + n_3)}$$

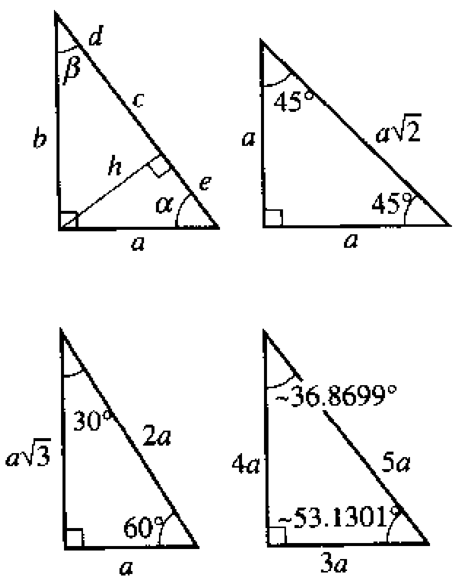
5. 高线的交点



$$C_{alt} = \frac{n_1V_1 + n_2V_2 + n_3V_3}{n_1 + n_2 + n_3}$$

C.2.3　直角三角形

下面列出了一些常用的直角三角形：



$$c^2 = a^2 + b^2$$

$$A = ab/2$$

$$h = ab/c$$

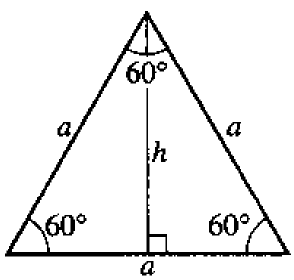
$$d = a^2/c$$

$$e = b^2/c$$

$$R = c/2$$

$$r = \frac{a + b - c}{2}$$

C.2.4　等边三角形



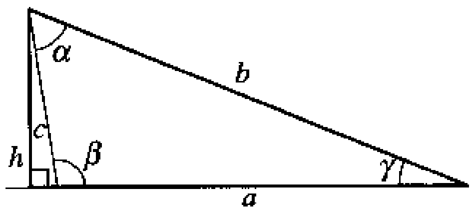
$$A = \frac{a^2\sqrt{3}}{4} = \frac{h^2}{\sqrt{3}}$$

$$h = \frac{a\sqrt{3}}{2}$$

$$R = \frac{a}{\sqrt{3}}$$

$$r = \frac{a}{2\sqrt{3}}$$

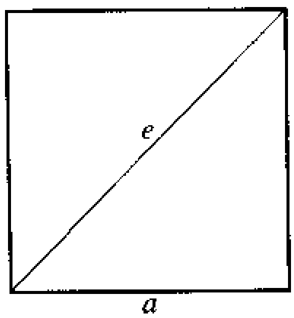
C.2.5　一般三角形



$$A = \frac{ah}{2}$$
$$= bc\sin\alpha$$
$$= \sqrt{p(p-a)(p-b)(p-c)}$$
$$h_a = c\sin\beta$$
$$= \frac{2\sqrt{p(p-a)(p-b)(p-c)}}{a}$$
$$m_a = \frac{1}{2}\sqrt{2b^2+2c^2-a^2}$$
$$s_a = \sqrt{bc[1-(\frac{a}{b+c})^2]}$$
$$R = \frac{abc}{4A}$$
$$r = \frac{2A}{a+b+c}$$
$$= \frac{A}{p}$$

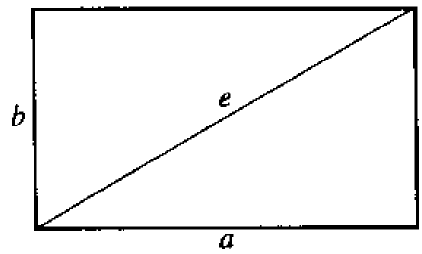
C.3　四边形

C.3.1　正方形



$$A = a^2$$
$$= \frac{e^2}{2}$$
$$R = \frac{a}{\sqrt{2}}$$
$$e = a\sqrt{2}$$
$$r = \frac{a}{2}$$

C.3.2　矩形

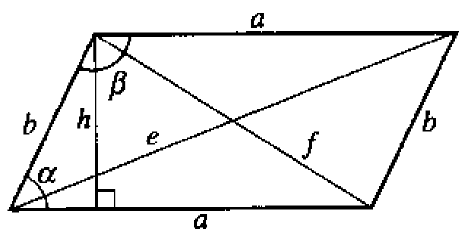


$$A = ab$$

$$R = \frac{e}{2}$$

$$e = \sqrt{a^2 + b^2}$$

C.3.3　平行四边形



$$A = ah$$

$$= a^2 \sin \alpha$$

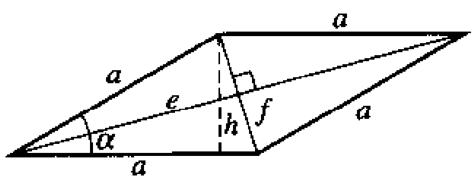
$$h = b \sin \alpha$$

$$e^2 + f^2 = 2(a^2 + b^2)$$

$$e = \sqrt{a^2 + b^2 + 2ab \cos \alpha}$$

$$e = \sqrt{a^2 + b^2 - 2ab \cos \alpha}$$

C.3.4　菱形



$$A = ah$$

$$= a^2 \sin \alpha$$

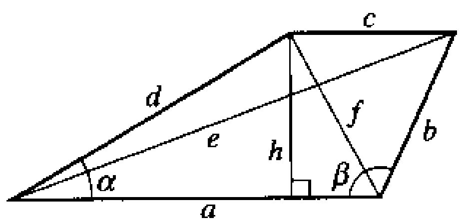
$$h = \frac{1}{2} ef$$

$$e^2 + f^2 = 4a^2$$

$$e = 2a \cos \frac{\alpha}{2}$$

$$f = 2a \sin \frac{\alpha}{2}$$

C.3.5　梯形



$$A = \frac{(a + c)h}{2}$$

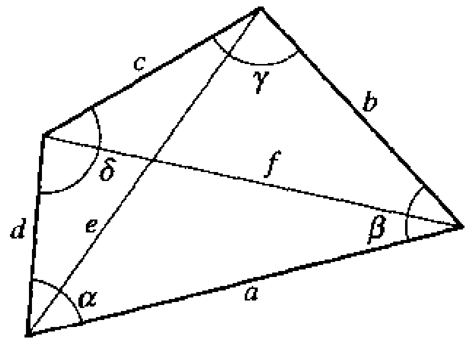
$$h = d \sin \alpha$$

$$= b \sin \beta$$

$$e = \sqrt{a^2 + b^2 - 2ab \cos \beta}$$

$$f = \sqrt{a^2 + d^2 - 2ad \cos \alpha}$$

C.3.6 一般四边形



$$\alpha+\beta+\delta+\gamma=360^{\circ}$$

$$\theta=90\Leftrightarrow a^2+c^2=b^2+d^2$$

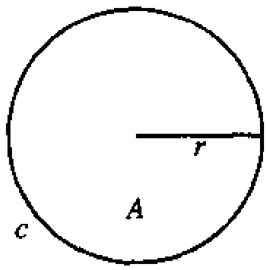
$$\begin{aligned}A&=\frac{1}{2}ef\sin\theta\\&=\frac{1}{4}(b^2+d^2-a^2-c^2)\tan\theta\\&=\frac{1}{4}\sqrt{4e^2f^2-(b^2+d^2-a^2-c^2)^2}\end{aligned}$$

C.4 圆

C.4.1 符号

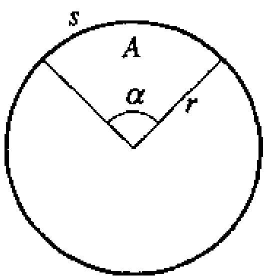
- r：半径
- d：直径
- c：周长
- s：弧长

C.4.2 完整的圆



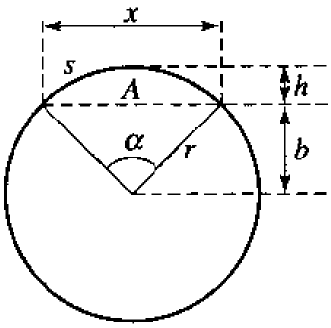
$$\begin{aligned}c&=2\pi r\\&=\pi d\\A&=\pi r^2\\&=\frac{\pi d^2}{4}\end{aligned}$$

C.4.3 扇形



$$\begin{aligned}s&=\alpha r\\A&=\frac{sr}{2}\\&=\frac{\alpha r^2}{2}\end{aligned}$$

C.4.4　圆的片断



$$x = 2r\sin\frac{\alpha}{2}$$

$$h = r\left(1 - \cos\frac{\alpha}{2}\right)$$

$$h(2r - h) = \left(\frac{x}{2}\right)^2$$

$$= \frac{\alpha r^2}{2}$$

$$A = \frac{r^2}{2}(\alpha - \sin\alpha)$$

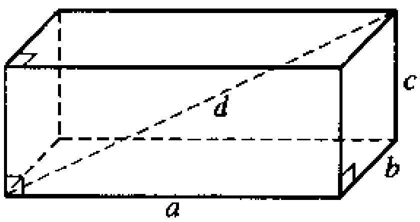
$$= \frac{1}{2}(rx - bx)$$

C.5　多面体

C.5.1　符号

- a，b，c：边
- d：对角线
- B：底面积
- S：表面积
- V：体积

C.5.2　箱体

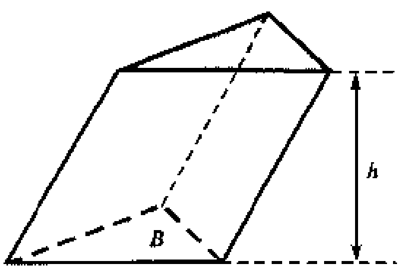


$$d = \sqrt{a^2 + b^2 + c^2}$$

$$S = 2(ab + bc + ac)$$

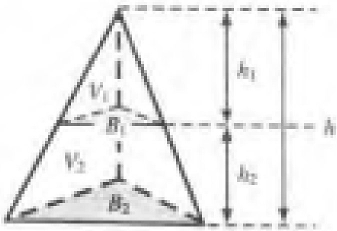
$$V = abc$$

C.5.3　棱柱



$$V = Bh$$

C.5.4 棱椎



$$V = \frac{1}{3}Bh$$

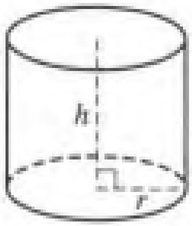
$$\frac{V_1}{V} = \left(\frac{B_1}{B}\right)^{\frac{3}{2}}$$

$$= \left(\frac{h_1}{h}\right)^3$$

$$V_1 = \frac{h_1^3 B}{3h^2}$$

$$V_2 = \frac{h_2}{3}\left(B + \sqrt{BB_1} + B_1\right)$$

C.6 圆柱



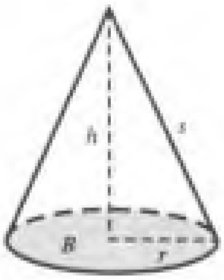
$$B = \pi r^2$$

$$A = 2\pi rh$$

$$S = 2\pi r(r + h)$$

$$V = \pi r^2 h$$

C.7 圆锥



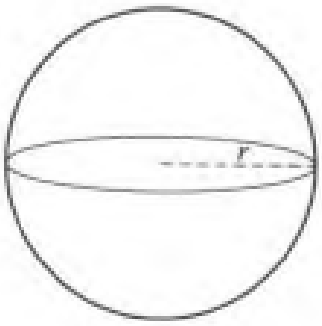
$$s = \sqrt{r^2 + h^2}$$

$$A = \pi rs$$

$$S = \pi r(r + s)$$

$$V = \frac{1}{3}\pi r^2 h$$

C.8 球

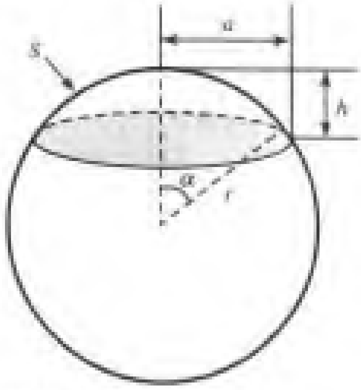


$$S = 4\pi r^2$$

$$V = \frac{4}{3}\pi r^3$$

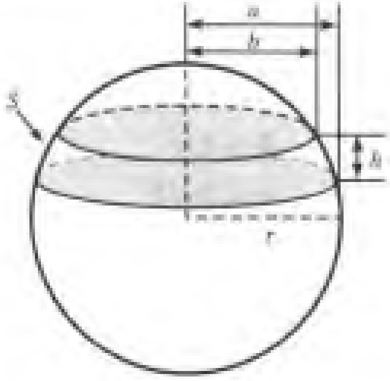
C.8.1 切片

1. 一个底的切片



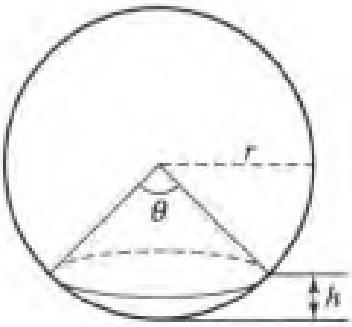
$$a = r\sin\alpha$$
$$a^2 = h(2r - h)$$
$$h = r(1 - \cos\alpha)$$
$$S = 2\pi rh$$
$$V = \frac{\pi}{3}h^2(3r - h) = \frac{\pi}{6}h(3a^2 + h^2)$$

2. 两个底的切片



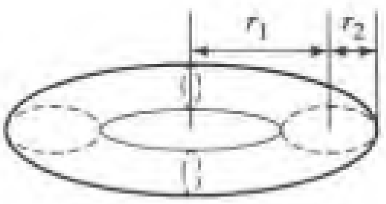
$$S = 2\pi rh$$
$$V = \frac{\pi}{6}h(3a^2 + 3b^2 + h^2)$$

C.8.2 扇区



$$V = \frac{2\pi r^2 h}{3} = \frac{\pi r^3}{3}\left(2 - 3\cos\frac{\theta}{2} + \cos^3\frac{\theta}{2}\right)$$

C.9 环面



$$S = 4\pi^2 r_1 r_2$$
$$V = 2\pi^2 r_1 r_2^{\,2}$$

参 考 文 献

Agnew, Jeanne, and Robert C. Knapp. 1978. *Linear Algebra with Applications*. Brooks/Cole Publishing Company, Monterey, CA.

Anton, Howard. 1980. *Calculus with Analytic Geometry*. John Wiley and Sons, New York.

Arvo, James. 1990. A simple method for box-sphere intersection testing. In Andrew Glassner, editor, *Graphics Gems*, Academic Press, New York, pages 335–339.

Arvo, James, editor. 1991. *Graphics Gems II*. Academic Press, San Diego.

Bajaj, C. L., C. M. Hoffman, J. E. H. Hopcroft, and R. E. Lynch. 1989. Tracing surface intersections. *Computer Aided Geometric Design*, 5:285–307.

Barequet, Gill, and Sariel Har-Peled. 1999. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. *Proc. 10th ACM-SIAM Sympos. Discrete Algorithms*, pages 82–91.

Barnhill, Robert E., and S. N. Kersey. 1990. A marching method for parametric surface/surface intersection. *Computer Aided Geometric Design*, 7:257–280.

Bartels, Richard H., John C. Beatty, and Brian A. Barsky. 1987. *An Introduction to Splines for Use in Computer Graphics and Geometric Modeling*. Morgan Kaufmann Publishers, San Francisco.

Bellman, R. E. 1987. *Dynamic Programming*. Princeton University Press, Princeton, NJ.

Blumenthal, Leonard M. 1970. *Theory and Applications of Distance Geometry*. Chelsea House Publishers, Broomall, PA.

Boeing Information & Support Services. 1997. DT_NURBS spline geometry subprogram library: Theory document, version 3.5. Carderock Division, Naval Surface Warfare Center.

Bourke, Paul. 1992. Intersection of a line and a sphere (or circle). *astronomy.swin.edu.au/pbourke/geometry/sphereline*.

Bowyer. A. 1981. Computing dirichlet tessellations. *The Computer Journal*, 24(2): 162–166.

Bowyer, Adrian, and John Woodwark. 1983. *A Programmer's Geometry*. Butterworth's, London.

Busboom, Axel, and Robert J. Schalkoff. 1996. Active stereo vision and direct surface parameter estimation: Curve-to-curve image plane mappings. *IEEE Pro-*

ceedings on Vision, Image, and Signal Processing, 143(2), April. Web version: *ece.clemson.edu/iaal/vsip1rw/vsip1rw.htm.*

Bykat, A. 1978. Convex hull of a finite set of points in two dimensions. *Information Processing Letters*, 7:296–298.

Cameron, S. 1997. Enhancing GJK: Computing minimum and penetration distances between convex polyhedra. *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3112–3117.

Cameron, S., and R. K. Culley. 1986. Determining the minimum translational distance between convex polyhedra. *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 591–596.

Campagna, Swen, Philipp Slusallek, and Hans-Peter Seidel. 1997. Ray tracing of spline surfaces: Bézier clipping, Chebyshev boxing, and bounding volume hierarchy—a critical comparison with new results. *The Visual Computer*, 13.

Casselman, Bill. 2001. Mathematics 309—conic sections and their applications. *www.math.ubc.ca/people/faculty/cass/courses/m309-01a/text/ch4.pdf.*

Chasen, Sylvan H. 1978. *Geometric Principles and Procedures for Computer Graphics Applications*. Prentice Hall, Englewood Cliffs, NJ.

Chazelle, B. 1991. Triangulating a simple polygon in linear time. *Disc. Comp. Geom.*, 6:485–524.

Chazelle, B., and J. Incerpi. 1984. Triangulation and shape complexity. *ACM Transactions on Graphics*, 3:135–152.

Chazelle, B., and L. Palios. 1990. Triangulating a nonconvex polyhedron. *Discrete and Computational Geometry*, 5:505–526.

Cheng, S.-W., T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. 2000. Sliver exudation. *Journal of the ACM*, 47(5):883–904.

Clarkson, K., R. E. Tarjan, and C. J. Van Wyk. 1989. A fast Las Vegas algorithm for triangulating a simple polygon. *Disc. Comp. Geom.*, 4:387–421.

Cohen, Elaine, Tom Lyche, and Richard Riesenfeld. 1980. Discrete B-splines and subdivision techniques in computer-aided geometric design and computer graphics. *Computer Graphics and Image Processing*, 14:87–111.

Cohen, Elaine, Richard F. Riesenfeld, and Gershon Elber. 2001. *Geometric Modeling with Splines: An Introduction*. A. K. Peters, Natick, MA.

Cohen, J. D., M. C. Lin, D. Manocha, and M. K. Ponamgi. 1995. I–Collide: An interactive and exact collision detection system for large-scale environments. *Proc. ACM Symposium on Interactive 3D Graphics*, pages 189–196.

Collins, G. E., and A. G. Akritas. 1976. Polynomial real root isolation using Descartes' rule of signs. *ACM Symposium on Symbolic and Algebraic Computation*, pages 272–276.

Collins, G. E., and R. Loos. 1982. Real zeros of polynomials. *Computing, Suppl.*, 4:83–94.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. The MIT Press, Cambridge, MA.

Crawford, Diane, editor. 2002. Game engines in scientific research (seven articles). *Communications of the ACM*, 45(1), January.

Dahmen, W., C. A. Micchelli, and H.-P. Seidel. 1992. Blossoming begets B-spline bases built better by B-patches. *Mathematics of Computation*, 1(1):97–115, July.

de Berg, Mark (editor), Marc van Kreveld, Mark Overmars, and O. Schwarzkopf. 2000. *Computational Geometry: Algorithms and Applications* (2nd edition). Springer, Berlin.

DeRose, Tony D. 1989. A coordinate-free approach to geometric programming. *Math for SIGGRAPH: Course Notes 23, SIGGRAPH '89*, pages 55–115, July.

DeRose, Tony D. 1992. *Three-Dimensional Computer Graphics: A Coordinate-Free Approach*. Unpublished manuscript, University of Washington.

Dey, Tamal K., Chandrajit L. Bajaj, and Kokicki Sugihara. 1991. On good triangulations in three dimensions. *Proceedings of the First Symposium on Solid Modeling Foundations and CAD/CAM Applications*, pages 431–441.

Dobkin, D. P., and D. G. Kirkpatrick. 1990. Determining the separation of preprocessed polyhedra—A unified approach. *Proc. 17th Internat. Colloq. Automata Lang. Program, Lecture Notes in Computer Science*, volume 443, pages 400–413. Springer-Verlag.

Dupont, Laurent, Sylvain Lazard, and Sylvain Petitjean. 2001. Towards the robust intersection of implicit quadrics. In *Workshop on Uncertainty in Geometric Computations, The University of Scheffield (England)*. Kluwer.

Eberly, David H. 1999. Polysolids and boolean operations. *www.magic-software.com/Documentation/psolid.pdf*.

Eberly, David H. 2000. *3D Game Engine Design*. Morgan Kaufmann, San Francisco.

Eberly, David H. 2001. Polysolid and BSP-based Boolean polygon operations. *www.magic-software.com/ConstructivePlanarGeometry.html*.

Edelsbrunner, H., and R. Seidel. 1986. Voronoi diagrams and arrangements. *Disc. Comp. Geom.*, 1:25–44.

Farin, Gerald. 1990. *Curves and Surfaces in Computer Aided Geometric Design: A Practical Guide*. Academic Press, Boston.

Farin, Gerald. 1995. *NURB Curves and Surfaces, From Projective Geometry to Practical Use*. A. K. Peters, Wellesley, MA.

Farouki, R. T. 1986. The characterization of parametric surface sections. *Computer Vision, Graphics, and Image Processing*, 33:209–236.

Field, D. A. 1986. Implementing Watson's algorithm in three dimensions. *Proceedings of the Second Annual ACM SIGACT/SIGGRAPH Symposium on Computational Geometry*, pages 246–259.

Finney, Ross L., and George B. Thomas. 1996. *Calculus and Analytic Geometry*, 9th edition. Addison-Wesley Publishing Company, Reading, MA.

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes. 1996. *Computer Graphics: Principles and Practices*. Addison-Wesley Publishing Company, Reading, MA.

Fournier, Alain, and John Buchanan. 1984. Chebyshev polynomials for boxing and intersections of parametric curves and surfaces. *Computer Graphics Forum: Proceedings of Eurographics '94*, volume 13(3), pages 127–142.

Fournier, A., and D. Y. Montuno. 1984. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3:153–174.

Fuchs, Henry, Zvi Kedem, and Bruce Naylor. 1979. Predetermining visibility priority in 3-d scenes. *Proceedings of SIGGRAPH*, pages 175–181.

Fuchs, Henry, Zvi Kedem, and Bruce Naylor. 1980. On visible surface generation by a priori tree structures. *Proceedings of SIGGRAPH*, pages 124–133.

Gaertner, Bernd, and Sven Schoenherr. 1998. Exact primitives for smallest enclosing ellipses. *Information Processing Letters*, 68:33–38.

Gaertner, Bernd, and Sven Schoenherr. 2000. An efficient, exact, and generic quadratic programming solver for geometric optimization. *Proc. 16th Annual ACM Symposium on Computational Geometry (SCG)*, pages 110–118.

Georgiades, Príamos. 1992. Signed distance from point to plane. In David Kirk, editor, *Graphics Gems III*, Academic Press, New York, pages 223–224.

Gilbert, E. G., and C.-P. Foo. 1990. Computing the distance between general convex objects in three-dimensional space. *IEEE Transactions on Robotics and Automation*, 6(1):53–61.

Gilbert, E. G., D. W. Johnson, and S. S. Keerthi. 1988. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 4(2):193–203.

Glaeser, Georg. 1994. *Fast Algorithms for 3D-Graphics*. Springer-Verlag, New York.

Glassner, Andrew S., editor. 1989. *An Introduction to Ray Tracing*. Academic Press, Berkeley.

Glassner, Andrew S., editor. 1990. *Graphics Gems*. Academic Press, San Diego.

Goldman, Ronald N. 1985. Illicit expressions in vector algebra. *ACM Transactions on Graphics*, 4(3):223–243, July.

Goldman, Ronald N. 1987. Vector geometry: A coordinate-free approach. *Geometry for Computer Graphics and Computer Aided Design: Course Notes 19, SIGGRAPH '87*, pages 1–172, June.

Goldman, Ronald N. 1990a. Intersection of three planes. In Andrew Glassner, editor, *Graphics Gems*, Academic Press, San Diego, page 305.

Goldman, Ronald N. 1990b. Matrices and transformations. In Andrew Glassner, editor, *Graphics Gems*, Academic Press, San Diego, pages 472–475.

Goldman, Ronald N. 1990c. Triangles. In Andrew Glassner, editor, *Graphics Gems*, Academic Press, San Diego, pages 20–23.

Goldman, Ronald N. 1991. More matrices and transformations: Shear and pseudo-perspective. In James Arvo, editor, *Graphics Gems II*, Academic Press, San Diego, pages 338–341.

Goldman, Ronald N., and Tom Lyche, editors. 1993. *Knot Insertion and Deletion Algorithms for B–Spline Curves and Surfaces*. Society for Industrial and Applied Mathematics, Philadelphia.

Golub, Gene H., and Charles F. Van Loan. 1993. *Matrix Computations*, 2nd edition. The Johns Hopkins University Press, Baltimore, MD.

Gottschalk, Stefan, Ming Lin, and Dinesh Manocha. 1996. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics (SIGGRAPH '96 Proceedings)*, pages 171–180, August.

Haines, Eric. 1987. Abnormal normals. *Ray Tracing News*, 0, September.

Haines, Eric. 1989. Essential ray tracing algorithms. In Andrew Glassner, editor, *An Introduction to Ray Tracing*, Academic Press, San Diego, pages 33–77.

Haines, Eric. 1991. Fast ray–convex polyhedron intersection. In James Arvo, editor, *Graphics Gems II*, Academic Press, San Diego, pages 247–250.

Haines, Eric. 1994. Point in polygon strategies. In Paul S. Heckbert, editor, *Graphics Gems IV*, Academic Press, San Diego, pages 24–46.

Hanrahan, Pat. 1983. Ray tracing algebraic surfaces. *Computer Graphics (SIGGRAPH '83 Proceedings)*, ACM, July, pages 83–90.

Hart, John. 1994. Distance to an ellipsoid. In Paul S. Heckbert, editor, *Graphics Gems IV*, Academic Press, New York, pages 113–119.

Heckbert, Paul S., editor. 1994. *Graphics Gems IV*. Academic Press, San Diego.

Hecker, Chris. 1997. Physics, part 4: The third dimension. *Game Developer*, pages 15–26, June.

Hershberger, John E., and Jack S. Snoeyink. 1988. Erased decompositions of lines and convex decompositions of polyhedra. *Computational Geometry, Theory and Applications*, 9(3):129–143.

Hertel, S., and K. Mehlhorn. 1983. Fast triangulation of simple polygons. *Proc. 4th Internat. Conf. Found. Comput. Theory*, volume 158 of *Lecture Notes in Computer Science*, pages 207–218.

Hill, F. S., Jr. 1994. The pleasures of 'perp dot' products. In Paul S. Heckbert, editor,

Graphics Gems IV, Academic Press, New York, pages 139–148.

Hoffman, C. M. 1989. *Geometric and Solid Modeling*. Morgan Kaufmann, San Francisco.

Holwerda, Klaas. 2000. Boolean, version 6. *www.xs4all.nl/ kholwerd/bool.html.*

Horn, Roger A., and Charles R. Johnson. 1985. *Matrix Analysis*. Cambridge University Press, Cambridge, England.

Huber, Ernst H. 1998. Intersecting general parametric surfaces using bounding volumes. In Mike Soss, editor, *Proceedings of the 10th Canadian Conference on Computational Geometry*, Montréal, Québec, School of Computer Science, McGill University, pages 52–53.

Joe, Barry. 1991. Delaunay versus max-min solid angle triangulations for three-dimensional mesh generation. *International Journal for Numerical Methods in Engineering*, 31:987–997.

Johnson, L., and R. Riess. 1982. *Numerical Analysis*. Addison-Wesley Publishing Company, Reading, MA.

Kajiya, James T. 1982. Ray tracing parametric surfaces. *Computer Graphics (SIGGRAPH '82 Proceedings)*, ACM, volume 16(3), pages 245–254 .

Kay, D. D. 1988. *Schaum's Outline of Theory and Problems of Tensor Calculus*. McGraw-Hill, New York.

Kay, Timothy L., and James T. Kajiya. 1986. Ray tracing complex scenes. *Computer Graphics (SIGGRAPH '86 Proceedings)*, ACM, pages 269–278.

Keil, J. M. 1985. Decomposing a polygon into simpler components. *SIAM J. Comput.*, 14:799–817.

Keil, J. M., and J. Snoeyink. 1998. On the time bound for convex decomposition of simple polygons. *Proceedings of the 10th Canadian Conference on Computational Geometry*, pages 54–55.

Kirk, David, editor. 1992. *Graphics Gems III*. Academic Press, San Diego.

Kirkpatrick, D. G. 1983. Optimal search in planar subdivisions. *SIAM J. Comp.*, 12:28–35.

Klee, V. 1980. On the complexity of d-dimensional Voronoi diagrams. *Archiv. Mathem.*, 34:75–80.

Krishnan, Shankar, and Dinesh Manocha. 1997. An efficient surface intersection algorithm based on lower dimensional formulation. *ACM Transactions on Graphics*, 16(1).

Lam, T. 1973. *The Algebraic Theory of Quadratic Forms*. W.A. Benjamin, Reading, MA.

Lane, Jeffrey, and Robert F. Riesenfeld. 1980. A theoretical development for the computer generation and display of piecewise polynomial surfaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2(1):150–159.

Lasser, D. 1986. Intersection of parametric surfaces in the Bernstein-Bézier representation. *Computer-Aided Design*, 18(4):186–192.

Lee, Randy B., and David A. Fredricks. 1984. Special feature: Intersection of parametric surfaces and a plane. *IEEE Computer Graphics and Applications*, 4(8):48–51, August.

Leonov, Michael. 1997. poly_boolean. *woland.it.nsc.ru/ leonov/clipdoc.html.*

Levin, Joshua. 1976. A parametric algorithm for drawing pictures of solid objects composed of quadric surfaces. *Communications of the ACM*, 19(11):553–563, October.

Levin, Joshua. 1979. Mathematical models for determining the intersection of quadric surfaces. *Computer Graphics and Image Processing*, 11(1):73–87.

Levin, Joshua. 1980. *QUISP: A Computer Processor for the Design and Display of Quadric-Surface Bodies*. Ph.D. dissertation, Rensselaer Polytechnic Institute, Troy, NY.

Levine, Ron. 2000. Collisions of moving objects. GD algorithms list at *sourceforge.net*, November.

Lin, M. C., and J. F. Canny. 1991. A fast algorithm for incremental distance computation. *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 1008–1014.

Martin, William, Elaine Cohen, Russell Fish, and Peter Shirley. 2000. Practical ray tracing of trimmed NURBS surfaces. *Journal of Graphics Tools*, 5(1):27–52.

Maynard, Hugh, and Lucio Tavernini. 1984. Boolean operations on polysolids. Unpublished work. (See a summary of the work in [Eberly 1997].)

Meister, G. H. 1975. Polygons have ears. *Amer. Math. Monthly*, 82:648–651.

Miller, James R. 1987. Geometric approaches to nonplanar quadric surface intersection curves. *ACM Transactions on Graphics*, 6(4), October.

Miller, James R. 1999a. Applications of vector geometry for robustness and speed. *IEEE Computer Graphics and Applications*, 19(4):68–73, July.

Miller, James R. 1999b. Vector geometry for computer graphics. *IEEE Computer Graphics and Applications*, 19(3):66–73, May.

Miller, James R., and Ronald N. Goldman. 1992. Using tangent balls to find plane sections of natural quadrics. *IEEE Computer Graphics and Applications*, 16(2):68–82, March.

Miller, James R., and Ronald N. Goldman. 1993a. Detecting and calculating conic sections in the intersection of two natural quadric surfaces, Part I: Theoretical analysis. Technical Report TR-93-1, Department of Computer Science, University of Kansas.

Miller, James R., and Ronald N. Goldman. 1993b. Detecting and calculating conic sections in the intersection of two natural quadric surfaces, Part II: Geometric constructions for detection and calculation. Technical Report TR-93-2, Department of Computer Science, University of Kansas.

Miller, James R., and Ronald Goldman. 1995. Geometric algorithms for detecting and calculating all conic sections in the intersection of any two natural quadric surfaces. *Computer Vision, Graphics, and Image Processing*, 57(1):55–66, January.

Mirtich, B. 1997. V-clip: Fast and robust polyhedral collision detection. *ACM Transactions on Graphics*, 17(3):177–208.

Möller, Tomas. 1997. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30.

Möller, Tomas, and Eric Haines. 1999. *Real-Time Rendering*. A.K. Peters, Ltd., Natick, MA.

Möller, Tomas, and Ben Trumbore. 1997. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28.

Murtha, Alan. 2000. gpc (general polygon clipper library), version 2.31. *www.cs.man.ac.uk/aig/staff/alan/software/index.html*.

Naylor, B. 1990. SCULPT: An interactive solid modeling tool. *Proceedings of Graphics Interface '90*, pages 138–148, May.

Naylor, B. 1992. Interactive solid geometry via partitioning trees. *Proceedings of Graphics Interface '92*, pages 11–18, May.

Naylor, B., J. Amanatides, and W. Thibault. 1990. Merging BSP trees yields polyhedral set operations. *Proceedings of SIGGRAPH*, pages 115–124.

Newman, W., and R. Sproull. 1979. *Principles of Interactive Computer Graphics*, 2nd edition. McGraw-Hill, New York.

Nishita, Tomoyuki, Thomas W. Sederberg, and Masanori Kakimoto. 1990. Ray tracing trimmed rational surface patches. *Computer Graphics (SIGGRAPH '90 Proceedings)*, ACM, volume 24 (4), pages 337–345.

O'Rourke, J. 1985. Finding minimal enclosing boxes. *Internat. J. Comput. Inform. Sci.*, 14:183–199, June.

O'Rourke, Joseph. 1998. *Computational Geometry in C*, 2nd edition. Cambridge University Press, Cambridge, England.

Paeth, Alan W., editor. 1995. *Graphics Gems V*. Academic Press, San Diego.

Patrikalakis, N. M. 1993. Surface-to-surface intersections. *IEEE Computer Graphics and Applications*, 13(1):89–95.

Piegl, Les. 1989. Geometric method of intersecting natural quadrics represented in trimmed surface form. *Computer-Aided Design*, 13(1):89–95.

Piegl, Les, and Wayne Tiller. 1995. *The NURBS Book*. Springer-Verlag, Berlin.

Pierre, Donald A. 1986. *Optimization Theory with Applications*. Dover Publications, New York.

Pirzadeh, Hormoz. 1999. Rotating calipers home page. *www.cs.mcgill.ca/˜orm/rotcal.html*.

Pratt, M. J., and A. D. Geisow. 1986. Surface/surface intersection problems. In J. A. Gregory, editor, *The Mathematics of Surfaces*, volume 6, Clarendon Press, Oxford, pages 117–142.

Preparata, Franco P., and Michael Ian Shamos. 1985. *Computational Geometry: An Introduction*. Springer-Verlag, New York.

Press, W. H., B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. 1988. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, Cambridge, England.

Rade, Lennart, and Bertil Westergren. 1995. *Mathematics Handbook for Science and Engineering*. Birkhauser, Boston.

Rogers, David F. 2001. *An Introduction to NURBS with Historical Perspective*. Morgan Kaufmann Publishers, San Francisco.

Rogers, David F., and J. A. Adams. 1990. *Mathematical Elements for Computer Graphics*, 2nd edition. McGraw-Hill, New York.

Rossignac, J. R., and A. A. G. Requicha. 1987. Piecewise-circular curves for geometric modeling. *IBM Journal on Research and Development*, 31(3):39–45.

Roth, Scott. 1981. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144.

Rusin, Dave. 1995. General formula for the area of a polygon. *www.math.niu.edu/~rusin/known-math/95/greens*.

Salomon, David. 1999. *Computer Graphics and Geometric Modeling*. Springer-Verlag, New York.

Sarraga, R. F. 1983. Algebraic methods for intersection of quadric surfaces in GMSOLID. *Computer Vision, Graphics, and Image Processing*, 22(2):222–238, May.

Schneider, Philip J. 1990. A Bézier curve-based root finder. In Andrew Glassner, editor, *Graphics Gems*, Academic Press, San Diego, pages 408–415.

Schutte, Klamer. 1995. An edge labeling approach to concave polygon clipping. *www.ph.tn.tudelft.nl/People/klamer/clip.ps.gz*.

Sechrest, S., and D. Greenberg. 1981. A visible polygon reconstruction algorithm. *Comput. Graph.*, 15(3):17–26.

Sederberg, Thomas W. 1983. *Implicit and Parametric Curves and Surfaces*. Ph.D. thesis, Purdue University.

Sederberg, Thomas W. 1984. Ray tracing of Steiner patches. *Computer Graphics (SIGGRAPH '84 Proceedings)*, ACM, volume 18 (3), pages 159–164.

Sederberg, Thomas W., and Tomoyuki Nishita. 1991. Geometric Hermite approximation of surface patch intersection curves. *Computer Aided Geometric Design*, 8:97–114.

Seidel, R. 1991. A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons. *Computational*

Geometry: Theory and Applications, 1(1):51–64.

Shamos, Michael I. 1978. *Computational Geometry*. Ph.D. dissertation, Yale University.

Shewchuk, Jonathan Richard. 2000. Stabbing Delaunay tetrahedralizations. *www.cs.cmu.edu/ jrs/papers/stab.ps.*

Shoemake, Ken. 1987. Animating rotation with quaternion calculus. *ACM SIGGRAPH Course Notes 10: Computer Animation: 3-D Motion, Specification, and Control.*

Sunday, Dan. 2001a. Area of triangles and polygons (2d and 3d). *www.softsurfer.com.*

Sunday, Dan. 2001b. Distance between lines, segments, and the closest point of approach. *www.softsurfer.com.*

Sunday, Dan. 2001c. Intersection of line, segment, and plane in 2d and 3d. *www.softsurfer.com.*

Sweeney, Michael A. J., and Richard H. Bartels. 1986. Ray tracing free-form B-spline surfaces. *IEEE Computer Graphics and Applications*, 6(2):41–49, February.

Tampieri, Filippo. 1992. Newell's method for computing the plane equation of a polygon. In David Kirk, editor, *Graphics Gems III*, Academic Press, San Diego, pages 231–232.

ter Haar Romeny, B. M., editor. 1994. *Geometry-Driven Diffusion in Computer Vision.* Computational Imaging and Vision Series. Kluwer Academic Publishers, Dordrecht, the Netherlands.

Thibault, William C., and Bruce F. Naylor. 1987. Set operations on polyhedra using binary space partitioning trees. *Proceedings of the 14th Annual Conference on Computer Graphics*, pages 153–162.

Toth, Daniel L. 1995. On ray tracing parametric surfaces. *Computer Graphics (SIGGRAPH '85 Proceedings)*, ACM, volume 19 (3), pages 171–179.

van den Bergen, Gino. 1997. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools*, 2(4):1–13.

van den Bergen, Gino. 1999. A fast and robust GJK implementation for collision detection of convex objects. *Journal of Graphics Tools*, 4(2):7–25.

van den Bergen, Gino. 2001a. Proximity queries and penetration depth computation on 3d game objects. *Game Developers Conference Proceedings*, pages 821–837.

van den Bergen, Gino. 2001b. SOLID: Software library for interference detection. *www.win.tue.nl/ gino/solid/.*

Vatti, B. R. 1992. A generic solution to polygon clipping. *Communications of the ACM*, 35(7):56–63.

Watson, D. 1981. Computing the n–dimensional Delaunay tessellation with applications to Voronoi polytopes. *The Computer Journal*, 24(2):167–172.

Wee, Chionh Eng, and Ronald N. Goldman. 1995a. Elimination and resultants part 1: Elimination and bivariate resultants. *IEEE Computer Graphics and Applications*,

January, pages 69–77.

Wee, Chionh Eng, and Ronald N. Goldman. 1995b. Elimination and resultants part 2: Multivariate resultants. *IEEE Computer Graphics and Applications*, March, pages 60–69.

Weiler, K., and P. Atherton. 1977. Hidden surface removal using polygon area sorting. *Proceedings of SIGGRAPH*, volume 11, pages 214–222.

Weisstein, Eric. 1999. Torus. *mathworld.wolfram.com/Torus.html.*

Welzl, Emo. 1991. Smallest enclosing disks (balls and ellipsoids). *Lecture Notes in Computer Science, New Results and New Trends in Computer Science*, 555:359–370.

Wikipedia. 2002. Field. *www.wikipedia.com/wiki/Field.*

Yamaguchi, Fujio. 1988. *Curves and Surfaces in Computer Aided Geometric Design.* Springer-Verlag, Berlin.

图索引

图 2.1　对示例的不明确变换的多种不同解释：（a）改变坐标；（b）平面内的变换；（c）从一个平面到另一个平面的变换……7
图 2.2　线性方程 $3x_1 + 2x_2 = 6$ 的解……18
图 2.3　两个方程的线性系统的三组可能解……19
图 2.4　函数的示意图……32
图 2.5　两个函数的组合……33
图 2.6　一对一映射、映成和同构映射……33
图 2.7　一对一函数和映成函数……34
图 2.8　一个可逆映射……34
图 2.9　最小二乘法示例……41
图 3.1　向量表示为有向线段……44
图 3.2　两个向量……45
图 3.3　首尾相接的两个向量……45
图 3.4　向量加法……45
图 3.5　向量加法链……46
图 3.6　向量减法……46
图 3.7　向量数乘……46
图 3.8　向量反转……47
图 3.9　向量加法的交换性……47
图 3.10　向量加法的结合性……47
图 3.11　乘法对加法的分配性……48
图 3.12　加法对乘法的分配性……48
图 3.13　三维空间中两个向量的生成空间是一个平面……49
图 3.14　一个以向量作为基底向量的线性组合……50
图 3.15　向量之间的夹角……51
图 3.16　确定方向的右手法则……52
图 3.17　一个向量表示为两组不同的基底向量的线性组合……53
图 3.18　正弦函数……54
图 3.19　线性变换“放大两倍”……54
图 3.20　非均衡缩放线性变换……55
图 3.21　旋转变换……55
图 3.22　剪切变换……55
图 3.23　点的减法……56
图 3.24　“首尾相接”公理……56
图 3.25　两个点的仿射组合……58

图 3.26　多点的仿射组合 59
图 3.27　向量之间的角度和向量的长度 60
图 3.28　向量加法的平行四边形法则 60
图 3.29　向量投影 61
图 3.30　$\cos\theta$ 为负 62
图 3.31　向量积 65
图 3.32　右手法则 65
图 3.33　三个向量定义的平行六面体 66
图 3.34　三重数积 66
图 3.35　一个仿射点相对于任意坐标系的坐标 68
图 3.36　仿射映射保持相对比例 69
图 3.37　向量和 71
图 3.38　向量缩放 71
图 3.39　点与向量的和 72
图 3.40　仿射映射的合成（旋转） 73
图 3.41　仿射坐标（a）和重心坐标（b） 73
图 3.42　最基本的三种单形：直线（a），三角形（b）和四面体（c） 75
图 4.1　$P = p_1\vec{\imath} + p_2\vec{\jmath} + p_3\vec{k} + \mathcal{O} = [p_1 \quad p_2 \quad p_3 \quad 1]$ 79
图 4.2　$\vec{v} = v_1\vec{\imath} + v_2\vec{\jmath} + v_3\vec{k} = [v_1 \quad v_2 \quad v_3 \quad 0]$ 79
图 4.3　垂直运算 85
图 4.4　垂直点积反映了向量之间的有符号角度 86
图 4.5　垂直点积与两个向量所构成的三角形的有符号面积相关 87
图 4.6　在$\mathcal{A}$和$\mathcal{B}$中表示P 90
图 4.7　在$\mathcal{G}$中表示$\mathcal{O}$ 91
图 4.8　在$\mathcal{B}$中表示$\vec{v}_i$ 91
图 4.9　平移 94
图 4.10　坐标系的平移 95
图 4.11　坐标系的简单旋转 96
图 4.12　一般旋转 98
图 4.13　显示在垂直于$\hat{u}$且包含P的平面$\mathcal{A}$上的一般旋转 99
图 4.14　缩放一个坐标系 100
图 4.15　均衡缩放 102
图 4.16　非均衡缩放 103
图 4.17　镜像 104
图 4.18　二维空间中的简单反射 105
图 4.19　三维空间中的简单反射 106
图 4.20　三维空间中的一般反射 106
图 4.21　二维空间中的镜像 107
图 4.22　三维空间中的镜像 108

图 4.23　二维空间中的剪切……108
图 4.24　$T_{xy,\theta}$……109
图 4.25　一般剪切说明……110
图 4.26　正射（正交）投影……112
图 4.27　斜轴投影……113
图 4.28　斜轴投影的切面……113
图 4.29　透视投影……115
图 4.30　交比……115
图 4.31　向量的透视映射……116
图 4.32　平面$x+y=k$……117
图 4.33　不正确的法线变换……117
图 4.34　法线表现为曲面切线的叉乘……118
图 5.1　（a）直线；（b）射线；（c）线段……121
图 5.2　直线的隐含定义……121
图 5.3　三角形的两种可能次序……123
图 5.4　三角形的参数形式的定义域和值域……123
图 5.5　三角形的重心形式的定义域和值域……124
图 5.6　矩形参数形式的定义域和值域……124
图 5.7　矩形的对称形式……125
图 5.8　一条典型的折线……125
图 5.9　示例：（a）简单的凹多边形；（b）简单的凸多边形……126
图 5.10　非简单多边形的示例：（a）交集不是顶点；
（b）交集是顶点。多边形是多体……126
图 5.11　折线链的例子：（a）严格单调；（b）单调，但不严格单调……126
图 5.12　一个单调多边形。正方形是一条链的顶点集；
三角形是另一条链的顶点集；圆是这两条链的顶点集……127
图 5.13　二次方程的解，取决于$d_0 \neq 0$，$d_1 \neq 0$和 r 的值……128
图 5.14　二次方程的解，取决于$d_0 \neq 0$，$d_1 = 0$，e_1和 r 的值……128
图 5.15　用距离（隐含）定义的圆及其参数形式……129
图 5.16　椭圆的定义……130
图 5.17　三次贝塞尔曲线……131
图 5.18　三次 B 样条曲线……132
图 6.1　一条直线上距指定点 Y 最近的点$X(\bar{t})$……134
图 6.2　一条射线上距指定点最近的点：（a）$X(\bar{t})$ 距 Y 最近；（b）P 距 Y 最近……135
图 6.3　一条线段上距指定点最近的点：（a）$X(\bar{t})$ 距 Y 最近；（b）P_0 距 Y 最近；
（c）P_1 距 Y 最近……135
图 6.4　从线段S_0中得到折线和 Y 之间的最小距离μ。S_1 和S_2 不能改变μ，
因为它们位于圆心为 Y、半径为μ的圆之外。线段S_3 不会改变μ，
因为它与该圆相交。无穷带测试不会排除 S_1和 S_3，因为它们部分地

同时位于两个无穷带中；然而，S_2被排除，因为它位于垂直带之外。
矩形测试排除S_1和S_2，因为它们位于包含该圆的矩形之外；然而，
S_3不会被排除……137
图 6.5 三角形上距给定点最近的点：（a）$\text{Dist}(Y, \mathcal{T}) = 0$；（b）$\text{Dist}(Y, \mathcal{T}) = \text{Dist}(Y, <P_0, P_1>)$；（c）$\text{Dist}(Y, \mathcal{T}) = \text{Dist}(Y, P_2)$；
（d）$\text{Dist}(Y, \mathcal{T}) = \text{Dist}(Y, <P_1, P_2>)$……139
图 6.6 将参数平面划分为 7 个区域……140
图 6.7 阶层曲线 $F(t_0, t_1)$ 与三角形的接触点：（a）与一条边相切；
（b）与一个顶点接触……140
图 6.8 阶层曲线 $F(t_0, t_1)$ 与三角形的接触点：（a）与一条边相切；
（b）与另一条边相切；（c）与一个顶点接触……141
图 6.9 一个三角形，一个矩形有界框和距它们的顶点和边最近的点域……146
图 6.10 一个矩形对平面的分区……151
图 6.11 （a）一个单锥的示例；（b）一个锥体的平截面；（c）一个正交平截面……152
图 6.12 在第一象限内的平截面部分……153
图 6.13 为寻找与测试点最近的点，只需搜索对测试点可见的边。
三条可见的边用点线表示。不可见的边用黑线表示。
可见的边位于一个以测试点为顶点、边线与该凸多边形相切的锥体内……154
图 6.14 二次曲线上距给定点最近的点……155
图 6.15 多项式曲线上距给定点最近的点……156
图 6.16 几种直线－直线构形：（a）零距离；（b）正距离……158
图 6.17 几种直线－射线构形：（a）零距离；（b）正距离……158
图 6.18 几种直线－线段构形：（a）零距离；（b）正距离……159
图 6.19 几种不平行的射线－射线构形：（a）零距离；
（b）从端点到内部点的正距离；（c）从端点到端点的正距离……160
图 6.20 阶层曲线 F 与位于$(\hat{t}_0, 0)$或（0，0）的最小边界的关系……161
图 6.21 几种平行的射线－射线构形：（a）射线指向同一方向；（b）射线指向
相反方向且并行重叠；（c）射线指向相反方向且不并行重叠……161
图 6.22 线段 S 获得当前最小距离 μ 的构形，μ 类似于图 6.4 中点 Y 获得的
当前最小距离……164
图 6.23 与最接近点相连的线段同时垂直于两个对象……165
图 6.24 （a）三角形 A 和 B；（b）集合$-B$；（c）集合 $A+B$；（d）集合 $A-B$，
其中灰色点是集合 $A-B$ 中距原点最近的点。黑色点是原点（0，0）……167
图 6.25 GJK 算法中的第一次迭代……168
图 6.26 GJK 算法中的第二次迭代……169
图 6.27 GJK 算法中的第三次迭代……169
图 6.28 GJK 算法中的第四次迭代……169
图 6.29 （a）凸包 $S_k \cup \{W_k\}$中 V_{k+1} 的图解；
（b）从 $M = \{W_0, W_2, W_3\}$中生成的新单形$\bar{S}_{k+1}$……170

图 7.1　圆沿逆时针方向从A到B形成的一段弧。包含A和B的直线将圆分为两部分，即该弧和圆的剩余部分。点P在该弧上，因为它与弧位于直线的同一边。点Q不在弧上，因为它位于直线的另一边……176
图 7.2　直线与三次曲线的相交……177
图 7.3　用分级外接界框检测直线－曲线的相交……179
图 7.4　在起始于零的格子中被栅格化的一条直线和一条曲线。直线使用掩码为 1（浅灰色）的或运算被栅格化。曲线使用掩码为 2（深灰色）的或运算被栅格化。同时容纳直线和曲线的格子单元的值为 3（点状）……180
图 7.5　椭圆和双曲线相交……182
图 7.6　两个圆的关系，$\vec{u} = C_1 - C_0$: (a) $\|\vec{u}\| = |r_0 + r_1|$; (b) $\|\vec{u}\| = |r_0 - r_1|$；(c) $|r_0 - r_1| < \|\vec{u}\| < |r_0 + r_1|$……183
图 7.7　$\mathcal{E}_1$包含在$\mathcal{E}_0$中。$\mathcal{E}_0$对$\mathcal{E}_1$的阶层曲线的最大值λ_1是负数……184
图 7.8　$\mathcal{E}_1$横切地与$\mathcal{E}_0$相交。$\mathcal{E}_0$对$\mathcal{E}_1$的阶层曲线的最小值λ_0是负数，最大值λ_1是正数……184
图 7.9　$\mathcal{E}_1$从$\mathcal{E}_0$中分离出来。$\mathcal{E}_0$对$\mathcal{E}_1$的阶层曲线的最小值λ_0是正数……185
图 7.10　两个椭圆的相交……186
图 7.11　在起始于零的格子中被栅格化的两条曲线。第一条曲线使用掩码为 1（浅灰色）的或运算被栅格化。第二条曲线使用掩码为 2（深灰色）的或运算被栅格化。同时容纳两条曲线的格子单元的值为 3（点状）……187
图 7.12　不相交的凸对象和它们的一条分离线……189
图 7.13　（a）不相交的凸多边形（b）相交的凸多边形……189
图 7.14　（a）边－边接触（b）顶点－边接触（c）顶点－顶点接触……190
图 7.15　距离非边法线的分离方向最近的边的法线：（a）来自同一个三角形；（b）来自不同的三角形……190
图 7.16　被第一个多边形的一条边的法线方向分离的两个多边形……192
图 7.17　（a）边－边相交预测（b）顶点－顶点相交预测（c）不相交预测……197
图 7.18　两个运动多边形的边－边接触……201
图 8.1　三点确定的圆……204
图 8.2　与三条直线相切的圆……204
图 8.3　与圆相切于给定点的直线……205
图 8.4　通过一个点并与圆相切的直线……206
图 8.5　一般有两条切线，但也可能只有一条切线，或者没有切线……206
图 8.6　与两个圆相切的直线……208
图 8.7　两圆之间的切线数目随圆的相对大小和位置不同而变化……209
图 8.8　两点和给定半径决定的圆……213
图 8.9　两点和给定半径决定的两个圆……213
图 8.10　求解由两点和给定半径决定的圆……213
图 8.11　经过一点并与一条直线相切且具有给定半径的圆……214
图 8.12　一般有两个不同的圆经过给定点……214

图 8.13　如果 P 在直线上，两个圆将以直线为轴成镜像；
　　如果 P 与直线的距离大于圆的直径，则无解……214
图 8.14　与两条直线相切并具有给定半径的圆……217
图 8.15　一般存在 4 个具有给定半径并与两条直线相切的圆……217
图 8.16　与两条直线相切的圆的求解方法……217
图 8.17　经过一点并与一个圆相切且具有给定半径的圆……219
图 8.18　可能存在四个圆、两个圆或无解，随圆的相对位置和半径的不同而变化……219
图 8.19　更具解析性的解法……219
图 8.20　更具实用性的方法……221
图 8.21　求解问题的实用性方法的特例……221
图 8.22　具有给定半径并与一条直线和一个圆相切的圆……222
图 8.23　解的数目随直线和圆的相对位置的不同而变化……222
图 8.24　如果给定的半径太小，则无解……223
图 8.25　求解具有给定半径的圆的解析……223
图 8.26　解的示意图……223
图 8.27　具有给定半径并与两圆相切的圆……225
图 8.28　一般存在两个解，但是解的数量随给定圆的相对大小和位置的不同而变化……226
图 8.29　与两圆相切的圆的求解……226
图 8.30　与一条给定直线垂直并通过一个给定点的直线……228
图 8.31　位于两点之间并与该两点等距的直线……228
图 8.32　与一条给定直线平行且相距指定值的直线……229
图 8.33　与一条给定直线平行且垂直或水平相距为指定值的直线……231
图 8.34　与一个给定圆相切并与一条给定直线垂直的直线……232
图 9.1　将平面定义为满足 $\vec{n}\cdot(X-P)=0$ 的所有点 X 的集合……235
图 9.2　平面方程系数的几何意义……236
图 9.3　平面的参数表示法……237
图 9.4　三维空间中圆的参数表示法……240
图 9.5　一个凸多边形和其三角形的扇形分解……241
图 9.6　一个非凸多边形和其三角形的扇形分解……241
图 9.7　一个三角形网格……241
图 9.8　顶点、边和三角形不是网格，因为一个顶点是孤立的……241
图 9.9　顶点、边和三角形不是网格，因为一条边是孤立的……242
图 9.10　顶点、边和三角形不是网格，因为两个三角形互相贯通……242
图 9.11　包含一个四面体、但增加了一个多余的顶点使中心面下沉的多面体……243
图 9.12　一个不是多面体的多边形网格，因为它没有相连在一起。四面体和矩形
　　网格共用同一个顶点并没有使它们满足边－三角形连通性而相连在一起……243
图 9.13　一个不是多面体的多边形网格，因为它的三个面共用一条边……243
图 9.14　一个凸多面体，一个规则的十二面体……243
图 9.15　两个相邻三角形的顺序的 4 种可能构形……247

图 9.16　两条平行的边相连在一起的矩形形成了（a）一个圆柱带（可定向的）或（b）麦比乌斯带（不可定向的）……247
图 9.17　5 种柏拉图立体。从左到右：四面体、六面体、八面体、十二面体、二十面体……249
图 9.18　具有三个非零特征值的二次曲面……254
图 9.19　具有两个非零特征值的二次曲面……255
图 9.20　具有一个非零特征值的二次曲面……256
图 9.21　一个标准的环面……256
图 9.22　一条三次贝塞尔曲线……258
图 9.23　一条三次B样条曲线……259
图 9.24　一个双三次B样条曲面……261
图 9.25　一个三次三角形B样条曲面……261
图 9.26　一个均匀的双三次B样条曲面……262
图 10.1　直线与点之间的距离……265
图 10.2　Q在$\mathcal{L}$上的投影……265
图 10.3　线段与点之间的距离……266
图 10.4　使用半空间法加快点与折线之间的距离测试……267
图 10.5　例子：点与折线之间距离的排除……268
图 10.6　点与平面之间的距离……271
图 10.7　平面$\mathcal{P}$的边缘视图……271
图 10.8　点与三角形之间的距离。最近的点可能在三角形内（a）在三角形的边上（b）或是三角形的一个顶点……272
图 10.9　用三角形区域对平面st分区……273
图 10.10　几种不同的阶层曲线$Q(s,t)=V$……274
图 10.11　矩形的另一种定义……277
图 10.12　点与矩形之间的距离……277
图 10.13　用矩形对平面分区……278
图 10.14　点到多边形的距离……279
图 10.15　用投影到二维空间的方法求解三维空间点与多边形的距离……280
图 10.16　典型例子：距圆最近的点……282
图 10.17　投影为圆心时……282
图 10.18　当P投影在圆盘内时的最近点……283
图 10.19　当P投影在圆盘外时的最近点……283
图 10.20　点到多面体（四面体）的距离……284
图 10.21　点到有向有界箱的距离……285
图 10.22　计算点与 OBB 的距离……286
图 10.23　第一象限内平截体的分区……288
图 10.24　椭球面上 6 种可能的“最近点”……293
图 10.25　任意点到参数形式曲线的距离……294

图 10.26　任意点到参数形式曲面的距离……295
图 10.27　两条直线之间的距离……297
图 10.28　各种可能的线形对象组合之间的距离的定义域……299
图 10.29　定义域边界可见性的定义……300
图 10.30　定义域有 4 条边的各种情形……302
图 10.31　两条线段之间的距离……302
图 10.32　直线与射线之间的距离……304
图 10.33　直线与线段之间的距离……305
图 10.34　两条射线之间的距离……307
图 10.35　射线和线段之间的距离……308
图 10.36　用单位正方形对 st 平面分区……311
图 10.37　几种不同的阶层曲线 $Q(s,t)=V$……311
图 10.38　直线与三角形之间的距离……317
图 10.39　三角形的参数表示形式……317
图 10.40　线形对象与三角形之间的距离问题的解空间的可能分区……318
图 10.41　区域 3 的边界带和边界平面……321
图 10.42　直线与矩形之间的距离……322
图 10.43　线段与矩形之间距离的定义域的可能分区……323
图 10.44　直线与四面体之间的距离……326
图 10.45　四面体先投影到（a）垂直于 $\hat{d}$ 的平面上，然后投影到（b）二维空间……327
图 10.46　直线与有向有界箱之间的距离……329
图 10.47　直线与 OBB 之间的距离的求解算法示意图……330
图 10.48　两个零分量的情形……331
图 10.49　一个零分量的情形……333
图 10.50　确定在何处寻找箱上的最近点……333
图 10.51　确定直线与箱是否相交……334
图 10.52　具有两条边和三个顶点的 OBB 的每个“正”面
都可能最接近于指定的直线……336
图 11.1　直线与平面的相交……353
图 11.2　直线与三角形的相交……355
图 11.3　射线与多边形的相交……358
图 11.4　线形对象与圆盘的相交……360
图 11.5　射线与多面体（八面体）的相交……361
图 11.6　线段与多边形（三角形）网格的相交……362
图 11.7　定义直线与多面体相交的半线的逻辑相交……363
图 11.8　如果直线与多面体不相交，则不存在半线的逻辑相交……363
图 11.9　射线与球面之间的可能相交……368
图 11.10　线形对象与椭球面的相交……369
图 11.11　圆柱面的参数标准表示法……372

图 11.12　圆柱面的一般表示法……372
图 11.13　圆锥面的参数标准表示法……376
图 11.14　圆锥面的一般表示法……376
图 11.15　一个锐角锥，内部区域用阴影表示……377
图 11.16　一个锐角双锥，内部区域用阴影表示……377
图 11.17　$c_2=0$ 的情形。(a) $c_0 \neq 0$；(b) $c_0=0$……378
图 11.18　射线与 NURBS 曲面的相交……381
图 11.19　由于曲面嵌片数量的不足导致相交计算的失败（为清楚起见，仅显示其横截面）……383
图 11.20　表示为两个平面的相交的一条射线……384
图 11.21　叶子节点有界箱由每一对精选的顶点之间的贝塞尔曲线多边形构建……387
图 11.22　相邻的有界箱组合成一个下一级的有界箱……387
图 11.23　两个平面之间的相交……389
图 11.24　表 11.1 中描述的三个平面的可能构形……390
图 11.25　平面一三角形相交……392
图 11.26　平面一三角形相交的构形……393
图 11.27　三角形一三角形相交的构形：(a) $\mathcal{P}_0 \| \mathcal{P}_1$，但 $\mathcal{P}_0 \neq \mathcal{P}_1$；(b) $\mathcal{P}_0 = \mathcal{P}_1$；(c) $\mathcal{T}_0$ 与 $\mathcal{T}_1$ 相交；(d) $\mathcal{T}_0$ 与 $\mathcal{T}_1$ 不相交……397
图 11.28　三角形一三角形区间重叠的构形：(a) 相交；(b) 不相交；(c) ？……397
图 11.29　三角网格与平面的相交……399
图 11.30　多边形与平面的相交……400
图 11.31　多边形与三角形的相交……401
图 11.32　平面与球面的相交……402
图 11.33　球面一平面相交的横截面视图……403
图 11.34　平面与圆柱面的相交……405
图 11.35　平面与圆柱面相交的几种方式……405
图 11.36　平面一圆柱面相交的边缘视图……407
图 11.37　三维空间中的椭圆……408
图 11.38　三维空间中的圆……408
图 11.39　丹德林构图……409
图 11.40　平面与圆柱面相交的横截面，两个球面用于定义相交的椭圆（据 Miller 和 Goldman，1992）……409
图 11.41　如果平面的法线与圆柱的轴平行，那么平面与圆柱相交为一个圆……412
图 11.42　平面与圆锥面的相交……414
图 11.43　平面与圆锥面相交的几种方式……414
图 11.44　平面与无限圆锥面的相交检测……415
图 11.45　平面一圆锥面相交的边缘视图……416
图 11.46　无限圆锥面的定义……418
图 11.47　双曲线和抛物线的几何定义……418

图 11.48　平面与圆锥面的抛物线相交（Miller 和 Goldman，1992）……419
图 11.49　平面与圆锥面的圆形相交（Miller 和 Goldman 1992）……422
图 11.50　平面与圆锥面的椭圆形相交（Miller 和 Goldman 1992）……423
图 11.51　平面与圆锥面的双曲线形相交（Miller 和 Goldman 1992）……424
图 11.52　平面与圆锥面的退化相交（Miller 和 Goldman 1992）……426
图 11.53　平面与参数形式曲面的相交……431
图 11.54　（三次）埃尔米特基函数……433
图 11.55　由端点和切线确定的三次埃尔米特曲线……433
图 11.56　参数空间中的一个子片映射到小片上的一个拓扑三角区域（Lee 和 Fredricks 1984）……434
图 11.57　三维空间中的相交曲线 $R(t)$……434
图 11.58　参数空间中的相交曲线 $R(t)$……435
图 11.59　两个 B 样条曲面的相交……447
图 11.60　曲面参数空间中的相交曲线……447
图 11.61　线形对象与轴对齐有界箱的相交……461
图 11.62　轴对齐有界箱表现为 3 块“厚板”的相交……462
图 11.63　用一块厚板裁剪一条直线……462
图 11.64　根据适当的厚板裁剪方式来计算射线—AABB 相交……464
图 11.65　定义一个有向有界箱……464
图 11.66　用“定向厚板”进行裁剪……465
图 11.67　平面与轴对齐有界箱的相交……468
图 11.68　我们只需检查与平面法线最平齐的对角线端点所在的角……468
图 11.69　平面与有向有界箱之间的相交……469
图 11.70　将一个 OBB 的对角线投影到平面的法线上……469
图 11.71　两个轴对齐有界箱之间的相交……470
图 11.72　OBB 相交检测的二维示意图（Gottschalk, Lin 和 Manocha 1996）……471
图 11.73　轴对齐有界箱与球面的相交……474
图 11.74　线形对象与环面的相交……485
图 11.75　计算环面上某一（相交）点的法线……486
图 11.76　环面上某一点的参数 u……486
图 11.77　环面上某一点的参数 v……487
图 12.1　点在平面上的投影……489
图 12.2　向量在平面上的投影……489
图 12.3　一个向量在另一个向量上的投影……490
图 12.4　直线与平面的夹角……490
图 12.5　两个平面之间的夹角……491
图 12.6　以一条直线为法线并通过一个点的平面……491
图 12.7　计算平面的距离系数……492
图 12.8　三点定义一个平面……493

图 12.9　3D 空间中两条直线所成的角度……493
图 12.10　3D 空间中两条直线所成的角度（反转其中一条直线的方向）……494
图 13.1　平面的空间分区二叉树分区……496
图 13.2　两条一致边具有相反方向的法线的分区线……497
图 13.3　构造一棵 BSP 树的多边形例子……498
图 13.4　处理边 ⟨9, 0⟩ 后的当前状态……499
图 13.5　处理边 ⟨0, 1⟩ 后的当前状态……499
图 13.6　处理边 ⟨1, 2⟩ 后的当前状态。该边使 ⟨4, 5⟩ 分裂为 ⟨4, 10⟩ 和 ⟨10, 5⟩，
也使 ⟨8, 9⟩ 分裂为 ⟨8, 11⟩ 和 ⟨11, 9⟩……499
图 13.7　处理边 ⟨10, 5⟩ 后的当前状态。该边使 ⟨7, 8⟩ 分裂为 ⟨7, 12⟩ 和 ⟨12, 8⟩……500
图 13.8　处理边 ⟨5, 6⟩ 后的当前状态……500
图 13.9　处理边 ⟨13, 9⟩ 后的最终状态……500
图 13.10　凸多边形的分区及相应的 BSP 树……501
图 13.11　凸多边形的分区及相应的平衡 BSP 树……502
图 13.12　线段分区……505
图 13.13　用通过测试点的垂直线来确定与该垂直线相交的两条边，
以此进行点在凸多边形内的测试。P 在多边形内。Q 在多边形外……515
图 13.14　用计算射线与多边形的交点的方法来进行点在多边形内的测试。
通过点 P_0 的射线仅横切通过多边形的边。交点个数是奇数（5），
因此该点在多边形内。通过点 P_1 的射线分析起来要复杂得多……516
图 13.15　类似 P 的在多边形“左边”的边上的点归类为在多边形内；
类似 Q 的在多边形“右边”的边上的点归类为在多边形外……518
图 13.16　图 13.14 中包含 P_1 的水平线的点的标记……519
图 13.17　图 13.14 中包含 P_1 的水平线的区间的标记……520
图 13.18　测试射线 $P + t\hat{d}$ 与共用边 $\vec{e}$ 相交于一个内部边上的点时的构形
（a）所有面在边和射线形成的平面的同一边。奇偶性没有变化
（b）所有面在边和射线形成的平面的不同边。奇偶性发生反转……524
图 13.19　球面多边形由共用测试射线相交的顶点 V 的边确定。
如果点 A 与射线的方向相符，则射线穿过多面体。
如果点 B 与射线的方向相符，则射线不穿过多面体……525
图 13.20　有界和无界多边形将平面分成内部和外部两个区域。内部区域显示为灰色。
右边的无界多边形是一个半空间，以一条单独的直线作为区域的边界……526
图 13.21　一个多边形与其反形。内部区域显示为灰色。边显示出
适当的方向使得内部总是在边的左边……526
图 13.22　两个内部区域是有界区域的多边形……527
图 13.23　两个多边形的交显示为灰色……527
图 13.24　两个多边形的并显示为灰色……527
图 13.25　两个多边形的差：（a）反 L 形多边形减去五边形。
（b）五边形减去反 L 形多边形……528

图 13.26　两个多边形的异或显示为灰色。该多边形是图 13.25 中显示的两个差的并……528
图 13.27　两个三角形的交：(a) 两个三角形 A 和 B (b) A 的相交部分的边在 B 内
(c) B 的相交部分的边在 A 内 (d) $A \cap B$ 是所有相交的边的总和……530
图 13.28　(a) 伪码报告不相交的两个多边形 (b) 实际的交集，一条线段……531
图 13.29　(a) 两个多边形 (b) 它们的真实交集……531
图 13.30　(a) 多边形有一个要求两组顶点/边的洞
(b) 可仅由一组顶点/边确定的锁眼……532
图 13.31　矩形和锁眼多边形的交……532
图 13.32　(a) 凸多边形 (b) 非凸多边形，因为连接 P 和 Q 的
线段并非完全位于源集内……536
图 13.33　一个点集与其凸包。除了成为凸包的顶点的点显示为黑色外，
其余点都显示为深灰色。包显示为浅灰色……537
图 13.34　凸包 H，H 之外的点 V，以及包的始于 V 的切线。
上下切点分别标为 P_U 和 P_L……538
图 13.35　P 与端点为 Q_0 和 Q_1 的线段的 5 种可能的关系：(a) P 在线段的左边；
(b) P 在线段的右边；(c) P 在线段所在的直线上，并在线段的左边；
(d) P 在线段所在的直线上，并在线段的右边；(e) P 在线段所在的
直线上，并在线段内……540
图 13.36　两个凸包 H_L 和 H_R 及其上下方的切线……545
图 13.37　两个凸包 H_L 和 H_R 及寻找较低的切线的增量式搜索……545
图 13.38　用于初始化切线搜索的极端点位于同一条垂直线上。虽然初始的可见性
测试都不会产生否定结果，但连接极端点的初始线段并不是包的切线。
当前的候选切线用点线显示……548
图 13.39　当前包和点将合并在一起。可见的面用浅灰色表示。隐藏面用深灰色表示。
分开这两个集合的折线用虚线显示。可见锥体的其他边用点线表示……549
图 13.40　(a) 两个二十面体。(b) 合并的包。虚线标识部分源包的面的边。
点线表示部分新增的面的边……551
图 13.41　(a) 棱椎和线段的侧视图。(b) 从线段后面观察的视图。线段 $\langle 0, a\rangle$
只能看见三角形 $\langle 2,3,6\rangle$ 和四边形 $\langle 3,4,5,6\rangle$。线段 $\langle a,b\rangle$ 只能看见四边形。
线段 $\langle b,1\rangle$ 只能看见三角形 $\langle 2,4,5\rangle$ 和四边形。在上述各种情形中都隐藏
的面是三角形 $\langle 2,3,4\rangle$ 和 $\langle 2,5,6\rangle$。包含这些三角形的边、一组线段的最终
图形是两个圆，而不是一个简单的圆……552
图 13.42　有限点集的三角剖分：(a) 包含可选要求；(b) 不包含可选要求……557
图 13.43　对一个凸四边形的两种三角剖分。角 $\alpha \doteq 0.46$ 弧度，角 $\beta \doteq 1.11$ 弧度。
(a) 顶部三角形的最小角是 α（小于 β）。(b) 最小角是 2α 弧度（小于 β）；
三角形使最小角最大化……557
图 13.44　图 13.43 中三角形的外接圆……557
图 13.45　(a) 新插入的点 P（显示为一个未标记的黑点）是三角形的内部点，这种情
形中三角形被分解为三个子三角形。(b) 该点位于三角形的一条边上，这种

情形中共用该边的每个三角形（如果存在的话）被分解为两个子三角形……558
图 13.46　一个需要交换边的三角形对(T, A)。为了处理网格的顶点－边－三角形表中的正确对象，必须跟踪它们的索引。交换边后，最多将产生两对三角形，即$\langle N_0, B_0\rangle$和$\langle N_1, B_1\rangle$，每对三角形都可能需要交换边。这些三角形对都要压入待处理的三角形对堆栈中……561
图 13.47　输入点集的超级三角形……562
图 13.48　包含将被插入的下一点的外接圆……563
图 13.49　针对将被插入的下一点的插入多边形……563
图 13.50　对整个网格恢复了空的外接圆条件而修正的插入多边形……563
图 13.51　最终的网格三角形显示为深灰色。删除了的三角形显示为浅灰色……564
图 13.52　（a）二维点的凸包提升为三维的抛物面。
（b）对应的德洛奈三角剖分，较低的包在xy平面上的投影……564
图 13.53　（a）互相可见的两个顶点。图中画出了连接它们的对角线。
（b）互相不可见的两个顶点，被它们之间的一个顶点阻挡。
（c）互相不可见的两个顶点，被它们之间的一条边阻挡。
（d）互相不可见的两个顶点，被多边形外的一个区域阻挡……565
图 13.54　图示说明为什么V_0与V_2之间缺少可见性等价于三角形$\langle V_0, V_1, V_2\rangle$包含一个优角顶点$R$……566
图 13.55　（a）凸角顶点的圆锥包容；（b）优角顶点的圆锥包容……567
图 13.56　用于说明水平梯形分解的简单多边形。边是随机标记的，并按数字的大小次序进行处理……572
图 13.57　整个平面是一个梯形……572
图 13.58　被$s_1.y_0$分解……573
图 13.59　被$s_1.y_1$分解……573
图 13.60　插入s_1……573
图 13.61　被$s_2.y_1$分解……573
图 13.62　被$s_2.y_0$分解……573
图 13.63　插入s_2……574
图 13.64　被$s_3.y_1$分解……574
图 13.65　插入s_3……574
图 13.66　插入s_9……575
图 13.67　梯形被合并到最大的梯形内之后的平面……577
图 13.68　梯形被合并到最大的梯形内之后的多边形例子……577
图 13.69　作为单调多边形的并的示例多边形。两个多边形分别显示为浅灰色和深灰色。梯形分解用的水平线仍然显示在图中……578
图 13.70　如果位于极值点的三角形是耳，则删除产生另一个单调多边形的耳……578
图 13.71　三角形$\langle V_0, V_{min}, V_1\rangle$成为一个耳的失败情形……579
图 13.72　（a）对W来说，并非所有优角顶点链上的顶点都是可见的。
（b）删除三角形导致W成为下一个增加到优角顶点链上的顶点……580

图 13.73 （a）W 出现在当前带的上方，V_0 对所有优角顶点链上的顶点都是可见的。
（b）删除三角形导致单调多边形的减少，因此处理能重复进行……580
图 13.74 （a）仅使用顶点进行分区。（b）使用多边形内的一个额外点进行分区……582
图 13.75 顶点 V_0 是优角顶点。对角线 $\langle V_0, V_1\rangle$ 是非基本对角线。
对角线 $\langle V_0, V_2\rangle$ 是 V_0 的基本对角线……583
图 13.76 原多边形（左上角）和 11 个最小凸分解，其中最窄对用灰色显示。
点线表示作为多边形的边而不是对角线来处理……584
图 13.77 多边形的最小凸分解中使用的凸多边形的规范三角剖分。
原多边形的边用粗线表示。分解中使用的对角线用点线表示。
用于规范三角剖分三角扇的对角线用细线表示……585
图 13.78 三角形的外接圆和内切圆……589
图 13.79 没有一致的多边形边的假设的最小面积矩形……593
图 13.80 （a）当前的有界圆和圆外的一个点，该点将使圆增大。（b）新的有界圆，
但原来在旧圆内的一个点现在位于新圆的外面，它将使算法重新开始执行…596
图 13.81 选择点 U_1 和 U_2 计算方程 13.2。三角形只有一条边对第一个点可见。
三角形的两条边对第二个点可见……604
图 A.1 顶部的次序显示首先进行一个不均匀比例缩放 $(x, y) \to (2x, y)$，
然后逆时针旋转 $\pi/4$ 弧度。底部的次序显示一次任意角度旋转
（圆在旋转时没有任何变化），但清楚地表明没有顶部次序中的
使圆变成椭圆的沿坐标轴的不均衡比例缩放……632
图 A.2 函数 $f(x, y) = z$ 和平面 $z = 0.8$ 相交产生一条阶层曲线，
显示为在 xy 平面上的投影……662
图 A.3 $f(x, y) = \frac{2x^2}{3} + y^2$ 表示的阶层曲线……662
图 A.4 $f(x, y) = \frac{2x^2}{3} + y^2$ 表示的阶层曲线在 xy 平面上的投影……663
图 A.5 无最小值或者最大值的两个函数……664
图 A.6 违背极值定理假设条件的两个函数……664
图 A.7 显示临界点的多个不同的函数。（a）和（b）是平稳点；
（c）和（d）是拐点……665
图 A.8 函数的最大值可能出现在区间的边界处或区间内部……666
图 A.9 $f(x) = x^3 + 6x^2 - 7x + 19, \forall x \in [-8, 3]$……666
图 A.10 $f'(x) = 3x^2 + 12x - 7, \forall x \in [-8, 3]$……667
图 A.11 相对极值……668
图 A.12 $f(x) = 3x^{\frac{5}{3}} - 15x^{\frac{2}{3}}$ 的相对极值……669
图 A.13 一个具有两个变量的函数的相对极值是其图形的峰和谷……669
图 A.14 函数 $z = f(x, y)$ 的相对极值（Anton，1980）……670
图 A.15 一个“鞍形函数”——点（0，0）不是一个极值点，
虽然第一个偏导数为 0……671
图 A.16 $2y^2x - yx^2 + 4xy$ 的图形，显示鞍点和相对极值点……673

图 A.17　$2y^2x - yx^2 + 4xy$的等高线图 ······673
图 A.18　椭圆$17x^2 + 8y^2 + 12xy = 100$的图形······676
图 A.19　函数$x^2 + y^2$的图形······676
图 A.20　$x^2 + y^2$的阶层曲线······677
图 A.21　约束曲线和椭圆相切于最小值点······677
图 A.22　球面$x^2 + y^2 + z^2 = 36$上距点（1，2，2）最近和最远的点······679
图 A.23　约束方程$g_1(x, y, z) = x - y + z = 1$和$g_2(x, y, z) = x^2 + y^2 = 1$······680
图 A.24　显示为由$g_1 = 0$和$g_2 = 0$所隐含定义的曲面相交所得的约束
图 A.24　曲线上点的f的极值点，以及f在这些极值点的水平集······681
图 B.1　角的标准术语······682
图 B.2　弧长的定义······683
图 B.3　弧度的定义······683
图 B.4　直角三角形的边的比值可用于定义三角函数······685
图 B.5　三角函数的一般定义······685
图 B.6　三角函数的几何解释······686
图 B.7　基础三角函数的图形······687
图 B.7　基础三角函数的图形（续）······688
图 B.8　正弦定理······692
图 B.9　正弦定理的证明······692
图 B.10　基础反三角函数的图形······699

表索引

表 2.1　本书使用的数学符号……10
表 6.1　使用内点到边的方法计算点到三角形距离的运算结果统计……145
表 6.2　使用边到内点方法计算点到三角形距离的运算结果统计……150
表 9.1　不同的柏拉图立体之间的关系……250
表 11.1　可用测试向量代数条件来区分的三个平面的 6 种可能构形……391
表 11.2　规范的简单约束二次曲面的参数表示法（Dupont, Lazard 和 Petitjean 2001）……440
表 11.3　投影二次曲面的参数表示法（Dupont, Lazard 和 Petitjean 2001）……440
表 11.4　自然二次曲面之间相交得到平面二次曲线的条件。据 Miller 和 Goldman (1995)……442
表 11.5　轴分解检测的系数……460
表 12.1　计算 ϕ 的公式……490
表 12.2　计算 θ 的公式……491
表 13.1　边与直线相交的标记是 o，i，m 和 p。本表用于更新位于相交点的当前标记。老的标记按行列出，与当前边相交的修正标记按列列出，与点相交的新的标记就是行与列对应的项……519
表 A.1　使用内存的比较……637
表 A.2　不同旋转表示法之间的转换运算结果的比较……638
表 A.3　变换一个向量的运算结果的比较……639
表 A.4　变换换 n 个向量的运算结果的比较……639
表 A.5　组合运算结果的比较……639
表 A.6　四元数插值运算结果……640
表 A.7　旋转矩阵插值运算结果……641
表 A.8　斯图姆多项式 t^3+3t^2-1 在 t 取不同的值时的符号……645
表 A.9　斯图姆多项式 $(t-1)^3$ 在 t 取不同的值时的符号……645
表 A.10　$f(x,y)=2y^2x-yx^2+4xy$ 在临界点的二阶偏微分……673
表 B.1　一些常用角度的三角函数值……686
表 B.2　三角函数的定义域和值域……687
表 B.3　反三角函数的定义域和值域……698